



HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Peer-to-Peer and Grid Computing

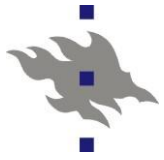
Chapter 5: Performance and Reliability of
Peer-to-Peer Systems





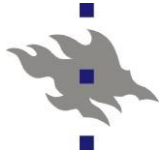
Chapter Outline

- n Cover performance and reliability issues in P2P systems
- n Evaluation of DHT performance
 - n Pure DHT performance issues
 - n Performance of DHT-based applications
- n Reliability issues in P2P systems
 - n Main focus on availability
- n Theoretical models of reliability
 - n How does replication improve reliability?
 - n How many copies do we need?
- n Load balancing issues with block-based systems



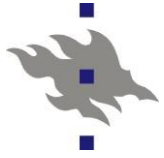
DHT Performance Issues

- n DHTs provide useful abstractions to programmers
- n What is the cost?
- n DHTs need to maintain overlay structure
 - n Additional communication needed
- n How should the parameters of a DHT be tuned?
 - n Number of successors, base, frequency of updates, etc.
- n Do DHTs maintain correctness in “normal” conditions?
 - n Most DHTs not evaluated against dynamic nodes
 - n What happens when lot of nodes join and leave?



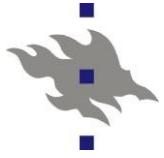
DHT Performance

- n How do DHTs cope with changes in membership?
- n How to compare different DHTs?
 - n How to figure out fundamental differences?
- n Most evaluations are about lookup latency or size of routing table in static networks
 - n Keeping large amount of state gives good results here!
 - n No penalty for large amount of state!
- n In normal conditions, periodic maintenance messages maintain overlay structure
 - n Compare DHTs in terms of how they maintain overlay
 - n Also include lookup performance
- n Comparisons done by Chord group at MIT
 - n Keep this in mind when looking at results!



Cost vs. Performance

- n Cost often measured as per-node state
- n More important metric: How to keep state up-to-date
 - n Up-to-date state avoids timeouts
 - n Also, need to find nearby neighbors
- n Cost metric: Number of bytes sent to network
 - n Network usually more limiting than CPU or memory
- n Performance metric: Lookup latency
 - n Dead nodes assumed to be detected quickly
 - n DHT retries other nodes after dead node
 - n Failed lookups converted to high latencies
 - Fair comparison?



Comparison

- n Compare 4 DHTS: Tapestry, Chord, Kademlia, and Kelips
- n Here we look at Chord and Tapestry only
- n Different parameters:

Tapestry

- n Base, stabilization interval, backup nodes

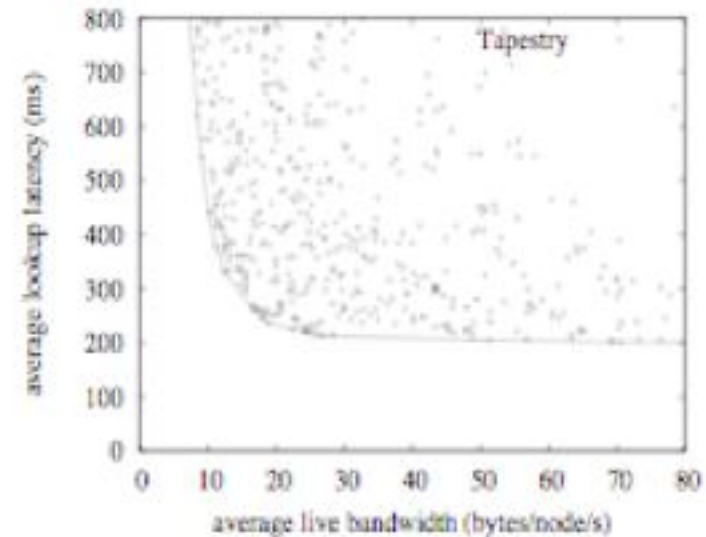
Chord

- n Number of successors, finger base



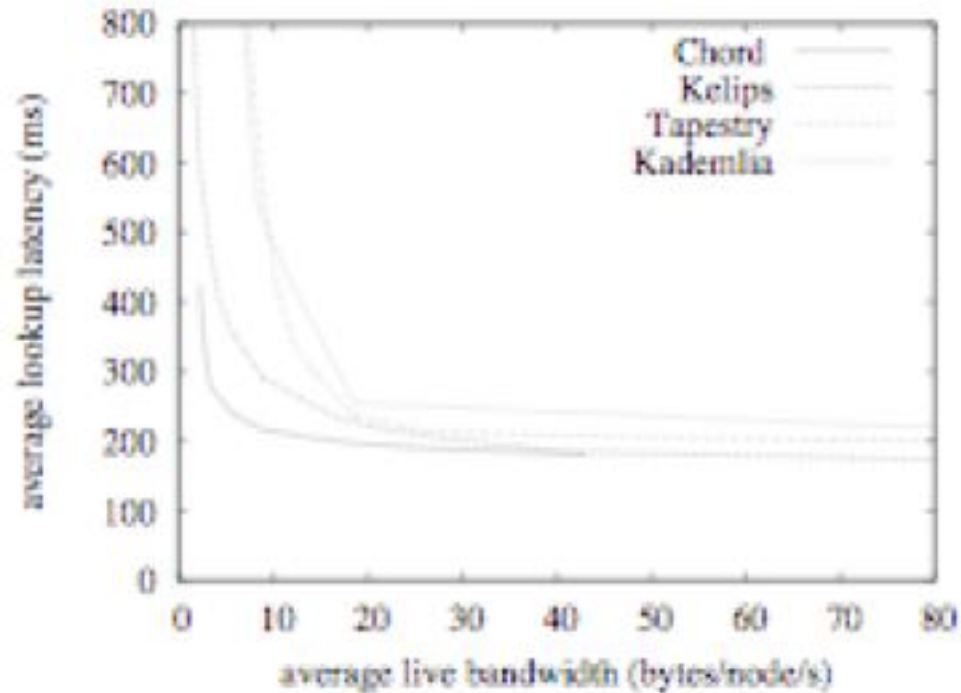
Evaluation Parameters

- n 1024 nodes in network
- n Only key lookup, no data retrieval
- n Nodes request random keys
 - n Exponentially, mean 10 mins
- n Nodes join and leave
 - n Exponentially, mean 1 hour
- n 6 hours simulated time
- n Nodes keep IP and ID
- n Many parameter combinations
- n **No single best choice**
- n Many optimal choices
- n Best points on *convex hull* of all points

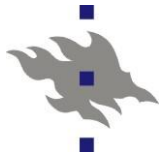




Overall Results

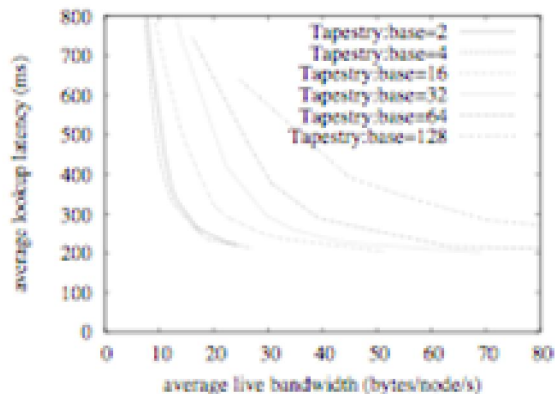


- n All 4 DHTs shown
- n Chord is “best”
 - n Kademia uses iterative routing, recursive appears to be better
- n Any DHT can be below 250ms latency
 - n Some need lot of bandwidth
- n Chord uses bandwidth efficiently
 - n Finger tables not needed
 - n Successor pointers maintain correctness, low bandwidth required
- n Other DHTs have no good correctness mechanisms

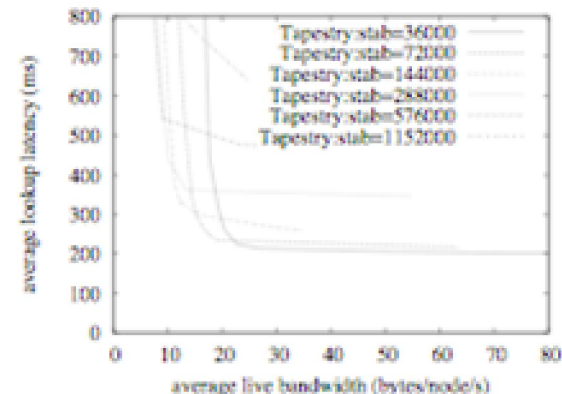


Tapestry: Effects of Parameters

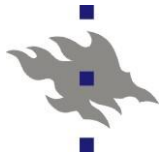
- As base decreases, less bandwidth is needed
 - Less entries in neighbor map, hence less traffic
- All bases can achieve same latency
 - Latency dominated by last hop, base can be small
- Stabilization can run frequently
 - Small increase in bandwidth, big reduction in latency



Base

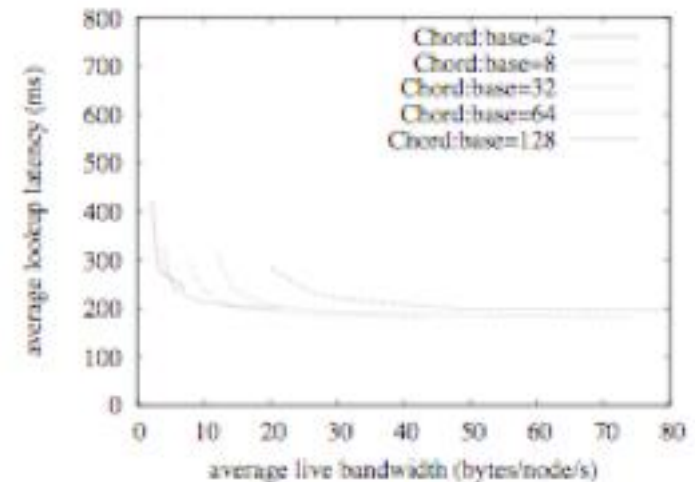


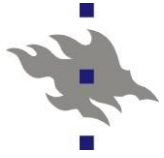
Stabilization interval



Chord: Effect of Parameters

- Chord only base is shown
 - Base is base of ID space
- No single best choice
- Convex hull is created by bases 2 and 8
- 72 second successor update interval is best (not shown)
 - Higher update wastes bandwidth
 - Lower update has more timeouts
 - Finger update interval affects only performance, can pick suitable value





DHT Performance: Summary

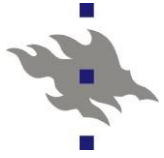
- n 4 different DHTs evaluated with different parameters
- n Cost of maintaining overlay vs. lookup latency

- n If tuned correctly, all 4 are about the same
- n Hard to tune correctly
 - n Parameters may interact
 - n Same parameter has different effects in different DHTs
 - n Some parameters are irrelevant



Performance of DHT-Based Applications

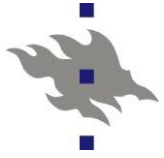
- n Above results show that we can configure a DHT to give us “decent” performance at “reasonable” cost
- n **Question:** Is “decent” good enough for real applications?
- n In other words, how does a DHT-based P2P application compare against a client/server-application?
- n **Recall:** Performance of CFS storage system in local network was about the same as FTP in wide area
- n How about other kinds of applications?
- n Let’s take Domain Name System (DNS) as example
 - n Fundamental Internet-service
 - n Very much a client/server application



P2P DNS

- n Domain Name System (DNS) very much client-server
- n *Ownership of domain = responsibility to serve its data*
- n DNS concentrates traffic on root servers
 - n Up to 18% of DNS traffic goes to root servers
- n Lot of traffic also due to misconfigurations

- n P2P DNS puts expertise in the system
 - n No need to be an expert administrator
- n P2P DNS shares load more equally
- n P2P DNS has much, much higher latencies :(

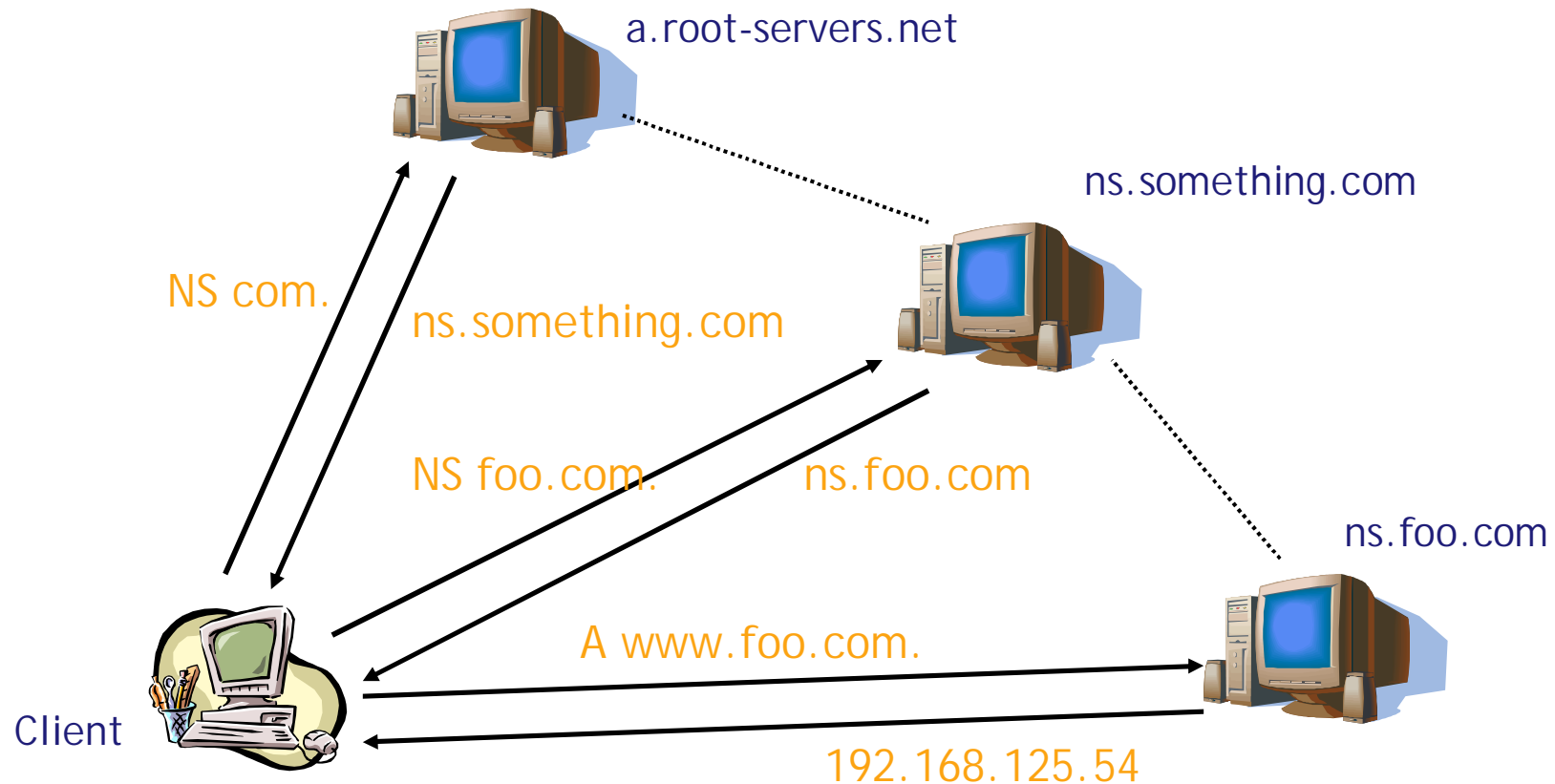


DNS: Overview

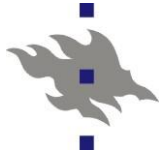
- n DNS organized in zones (\approx domain)
 - n Actual data in resource records (RR)
 - n Several types of RRs: A, PTR, NS, MX, CNAME, ...
- n Administrator of zone responsible for setting up a server for that zone (+ redundant servers at other domains)
- n Queries resolved hierarchically, starting from root
- n Owner of a zone is responsible for serving zone's data
- n DNS shortcomings:
 - n Need skill to configure name server
 - n No security (but added-on later to some degree)
 - n Queries can take very long in worst case



DNS: Example



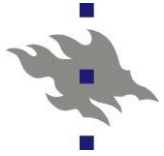
- n Client wants to resolve `www.foo.com`
- n Replies to queries have additional information (IP address + name)
- n Queries can be iterative (here) or recursive



How to Do P2P DNS?

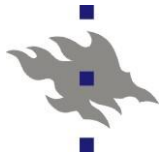
- n Put DNS resource records in a DHT
- n Key is hash of domain name and query type
 - n For example, SHA1(www.foo.com, A)
- n Values replicated on some replicas (~ 5-7)
- n Can be built on any DHT, works the same way
- n All resource records must be signed
 - n Some overhead for key retrieval

- n For migration, put P2P DNS server on local machine
 - n Configure normal DNS to go through P2P DNS
 - n No difference to applications



P2P DNS: Performance

- n Current DNS has median latency of 43 ms
 - n Measured at MIT
- n Some queries can take a long time
 - n Up to 1 minute (due to default timeouts)
- n P2P DNS has median latency of 350 ms!
 - n Simulated on top of Chord
- n P2P is much, much worse
 - n But extremely long queries cannot happen



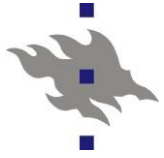
Why (Not) P2P DNS?

Pros

- n Simpler administration
 - n Most problems in current DNS are misconfigurations
 - n DNS servers not simple to configure well
- n P2P DNS robust against lost network connectivity
 - n Only outgoing link cut -> maybe not able to find own name
- n No risk of incorrect delegation
 - n Subdomains can be easily established
 - n Signatures confirm

Cons

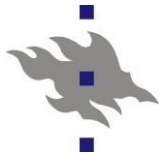
- n All queries must be anticipated in advance
 - n Not possible to set e.g. mail server for whole domain easily
- n Current DNS can tailor requests to client
 - n Widely used in content distribution networks and load balancing
- n Might be possible to implement above in client software



Future of DHT-Based Applications?

- n DHT-based applications have to make several RPCs
 - n 1 million node Chord = 20 RPCs, Tapestry 5 RPCs
- n Experiments with DNS show even 5 is too much
 - n Current DNS usually needs 2 RPCs
 - n DNS puts lot of knowledge at the top of the hierarchy
 - Root servers know about millions of domains
- n Many RPCs is main weakness of DHTs

- n DHT-based applications have all their features on clients
 - n New feature -> install new clients
 - n Some kind of an “active” network as a solution?

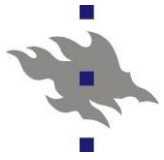


Reliability of P2P Storage

- n Example case: P2P storage system
 - n Each object replicated in some peers
 - n Peers can find where objects should be
 - Typically DHT-based, but DHT is not absolutely required
- n No concern of consistency
 - n Read-only storage system

Questions:

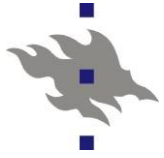
1. How many copies are needed for a given level of reliability?
 - n Unconstrained system with infinite resources
2. What is the optimal number of copies?
 - n System with storage constraints



Reliability of Data in DHT-Storage

- n Storage system using a distributed hash table (DHT)
- n Peer *A* wants to store object *O*
 - n Create *k* copies on different peers
 - n *k* peers determined by DHT for each object (*k* closest)
- n Later peer *B* wants to read *O*
 - n What can go wrong?

- n Simple storage system: Object created once, read many times, no modifications to object
- n Question: What is the value of *k* needed to achieve e.g., 99.9% availability of *O*?
 - n Remember: Only probabilistic guarantees possible!

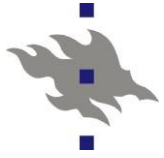


Assumptions

- n Assume I peers in the DHT
 - n Each peer has unlimited storage capacity
- n Peer is up with probability p
 - n Peers are homogeneous, i.e., all peers have same up-probability
- n Peers uniformly distributed in hash space
 - n Makes mathematical analysis tractable

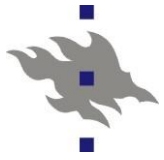
- n New peers can join the network
- n Peers never permanently leave

- n User may need to access several objects to complete one user-level action
 - n For example, resolve path name to file



What Can Go Wrong?

1. All k peers are down when B reads
 - Object is not available in any on-line peer
 2. Real k closest peers were down when A wrote and are up when B reads
 3. At least k peers join and become new closest peers
 - In above two cases, object is (maybe) still available in the peers where A wrote it
 4. All k peers have permanently left the network
 - Assumed not to happen
- n We look at only the first three cases
- n What are the probabilities of each one of them?



Probabilities of Loss

1. All k peers are down when B reads

$$p_{11} = (1 - p)^k$$

2. Real k closest peers were down when A wrote and are up when B reads

$$p_{12} \approx \sum_{i=k}^{(1-p)l} \binom{(1-p)l}{i} \left(\frac{p(1-p)}{l} \right)^i$$

3. N peers join and at least k peers become new closest peers

$$p_{13} = \sum_{i=k}^N \binom{N}{i} \frac{1}{l^i}$$



Numerical Values for Loss

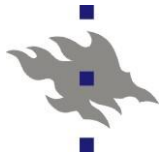
$$pl_3 \approx$$

I		
10^2	10^3	10^4
10^{-10}	10^{-15}	10^{-20}

p	$pl_1 \approx$
0.99	10^{-10}
0.9	10^{-8}
0.5	0.03
0.3	0.17

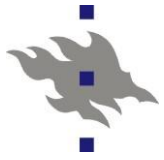
$pl_2 \approx$ (for given I and p)		
0	10^{-15}	10^{-15}
10^{-8}	10^{-8}	10^{-8}
10^{-4}	10^{-4}	10^{-4}
10^{-3}	10^{-3}	10^{-3}

- n First case (green) dominates clearly
 - n In above tables, $k = 5$
 - n For cases 2 and 3 also applies:
 - Search more than k nodes to find object



How to Improve?

- n Maintain storage invariant \rightarrow O always at k closest
 - n Needs additional coordination
 - n Possible if down-events controlled
 - n Crash \rightarrow others need to detect crash (before they crash)
 - n Guarantees availability as long as invariant maintained
 - n Possibly wastes storage if copies are not removed when peers come back into the system
 - n This approach taken by PAST storage system
- n Increase k
 - n Create more copies, simple to implement
 - n Wastes storage capacity?
 - n Not good for changing objects (consistency)



What the User Sees?

- Suppose: User's action needs to access several objects
 - For example, resolve path names of files one level at a time
- For each object: $p_s = 1 - p_{f_1} = 1 - (1 - p)^k$

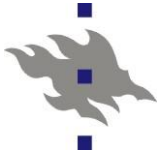
If we need to access 2 objects?

Success for user: $p_t = (1 - (1 - p)^k)^2$

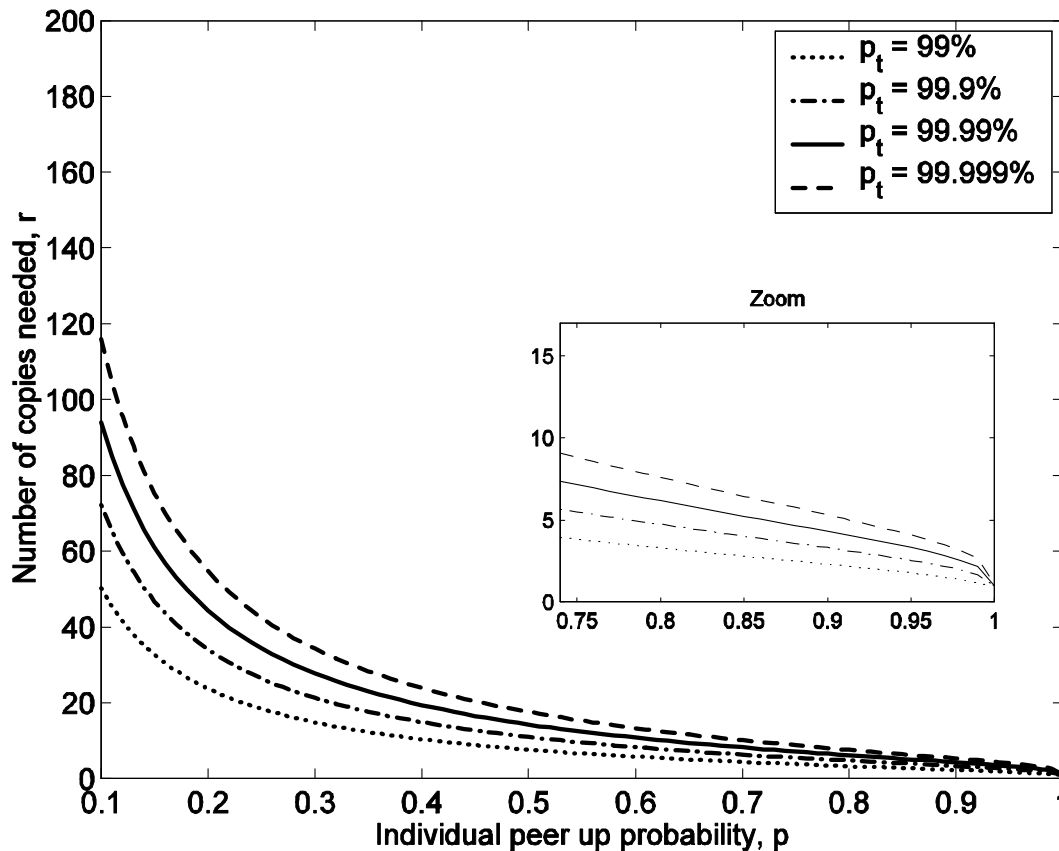
Solving for k :

$$k = \frac{\log(1 - \sqrt{p_t})}{\log(1 - p)}$$

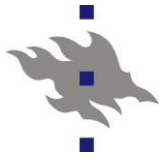
In general for n objects: $p_t = (1 - (1 - p)^k)^n$



How Large Should k Be?

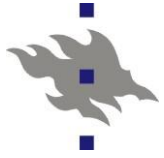


- Define target p_t
 - This is what user sees
 - Failures temporary
- When peers mostly up, k small
- Increase in p_t \rightarrow small increase in k



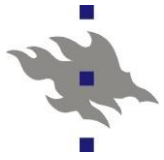
Replication Summary

- n Replication in read-only system helps availability
- n Main cause of unavailability is peers going down
- n Create k copies of each object
 - n If peers mostly up, k quite small (< 10)
 - n Maintaining actively copies in right peers helps
- n Above analysis assumes all objects equally popular or important
 - n Not always true
 - n Recall: Zipf-distribution for object popularities
 - n Also, some objects may require higher availability
- n How should objects be replicated in this case?

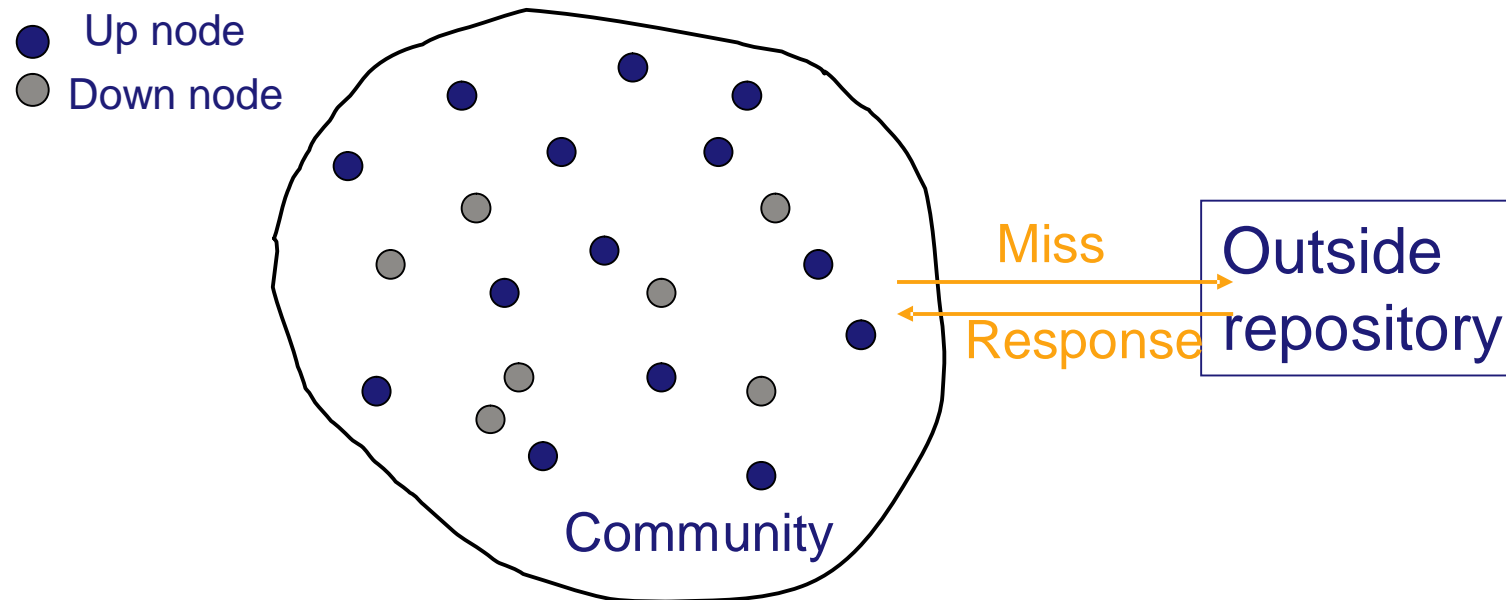


P2P Content Management

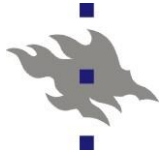
- n Group of peers access a set of files
 - n Some files are more popular than others
- n How many copies of each file should we have?
- n Where should the copies be placed?
- n Assumptions:
 - n DHT-based system for determining responsible nodes
 - n Set of files is static
 - n File popularities Zipf-distributed
- n P2P communities



Abstract Community Model



- Examples of communities: Campus, distribution engine
- Assume good bandwidth within community
- Goal: Satisfy requests from within community



Replication Issues

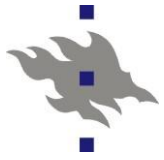
- n How many copies of each object in community?
- n Which peers in community have copies?
- n Is there an algorithm that is:
 - n simple
 - n decentralized
 - n adaptively replicates objects
 - n provides near-optimal replica profile?
- n What does “optimal replica profile” mean?



Replication Theory

- n J objects, I peers
- n object j
 - n requested with probability q_j
 - n size b_j
- n peer i
 - n up with probability p_i
 - n storage capacity S_i
- n decision variable
 - n $x_{ij} = 1$ if a replica of j is put in i ; 0 otherwise

- n Goal: maximize hit probability in community (availability)
- n Extension to byte hit probability is possible



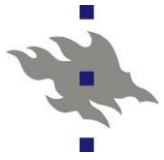
Optimization Problem

$$\text{Minimize } \sum_{j=1}^J q_j \prod_{i=1}^I (1 - p_i)^{x_{ij}}$$

$$\text{subject to } \sum_{j=1}^J b_j x_{ij} \leq S_i, \quad i = 1, \dots, I$$

$$x_{ij} \in \{0, 1\}, \quad i = 1, \dots, I, \quad j = 1, \dots, J$$

Can be reduced to Integer programming problem: NP



Homogeneous Up Probabilities

n Suppose $p_i = p$

n Let $n_j = \sum_{i=1}^I x_{ij}$ = number of replicas of object j

n Let S = total group storage capacity

n Minimize

$$\sum_{j=1}^J q_j (1-p)^{n_j}$$

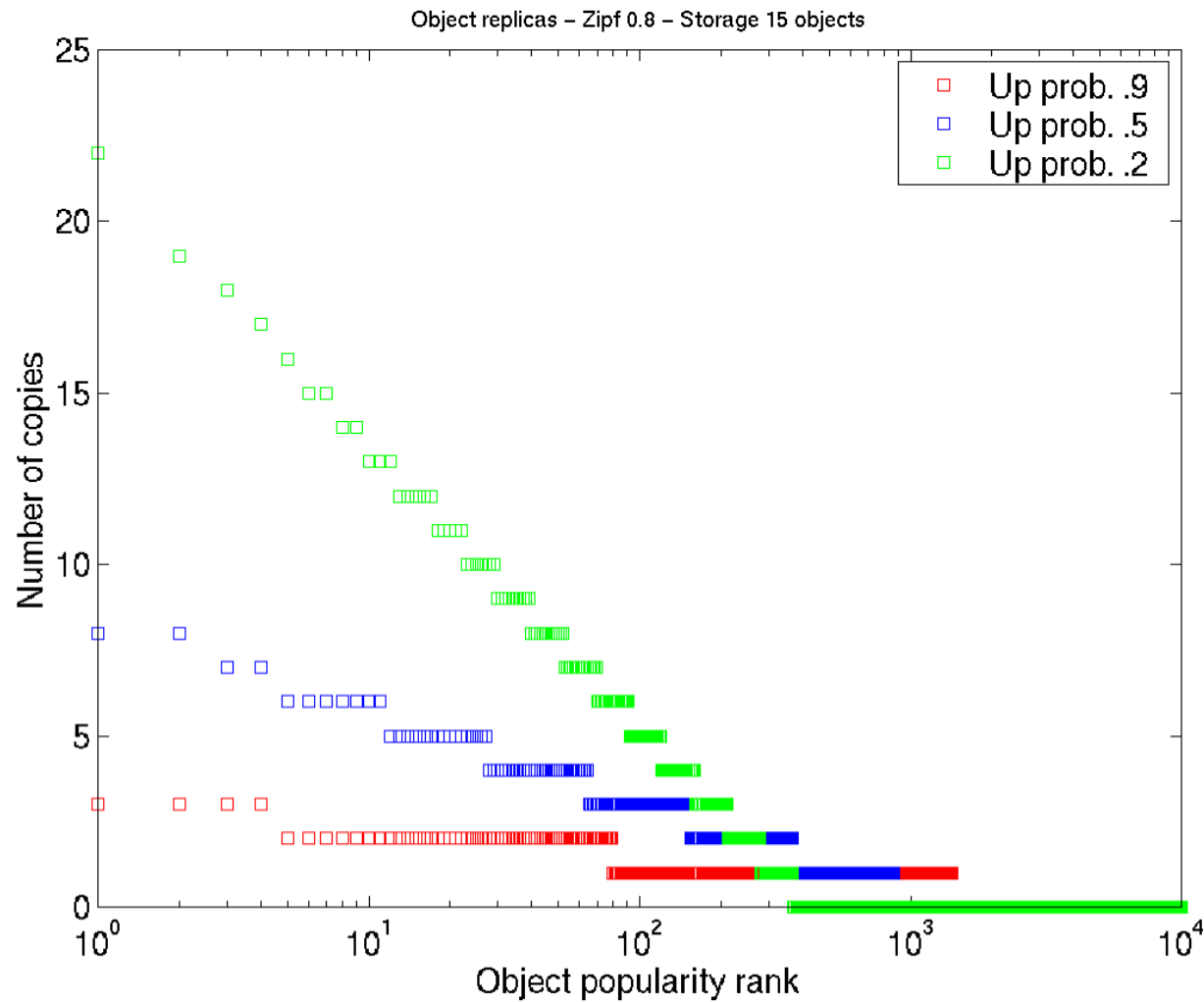
Can be solved by
dynamic programming

n subject to:

$$\sum_{j=1}^J b_j n_j \leq S$$



Replication vs. Up Probability



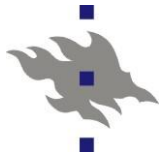
■ up prob = .9

■ up prob = .5

■ up prob = .2

Hit probabilities are different:

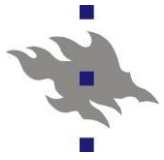
0.6, 0.4, 0.3



Problems with Optimal Solutions

- n Don't know *a priori* up/down probabilities
- n Don't know *a priori* object request rates
- n Object request rates are changing over time
- n New objects are being introduced

- n Need efficient adaptive algorithms!



Assumptions & Goals

Assume

- n Each object has a unique name (e.g., URN)
- n Each peer in community has shared and private storage
- n Each peer can access a DHT that gives current up winners for any object o

Goals

- n Replicate while satisfying requests (no extra work)
- n Adaptive, decentralized, simple
- n High availability: mimics optimal performance

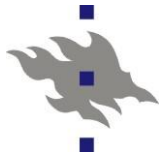


DHT: Winners

- Hash functions map each object name j into a “random” ordering of the nodes:

$$\text{hash}(j) \tilde{O} [i_j(1), i_j(2), \dots, i_j(l)]$$

- Each object j has a current “first-place winner,” “second-place winner,” etc.
- Winners are **current up winners**
- Any DHT can be modified to provide the winners

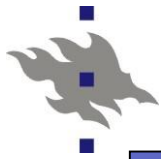


Adaptive Algorithm: Simple Version

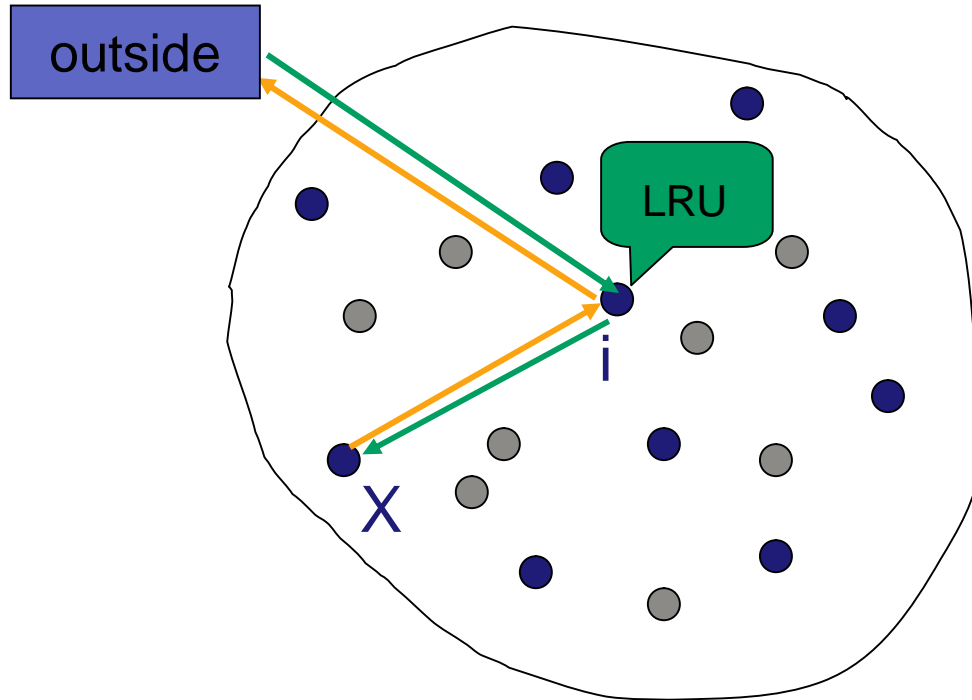
Suppose X is a node that wants object o .

- 1) X uses DHT to find 1st-place up node i for o
- 2) X asks i for o
- 3) If i doesn't have o , i retrieves o from the “outside” and stores a copy in its shared storage.
- 4) i sends o to X , which puts o in its private storage.

Each node uses LRU replacement policy in shared storage



Adaptive Algorithm



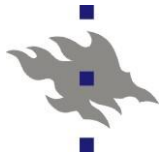
- up node
- down node

Each object o has “attractor nodes”

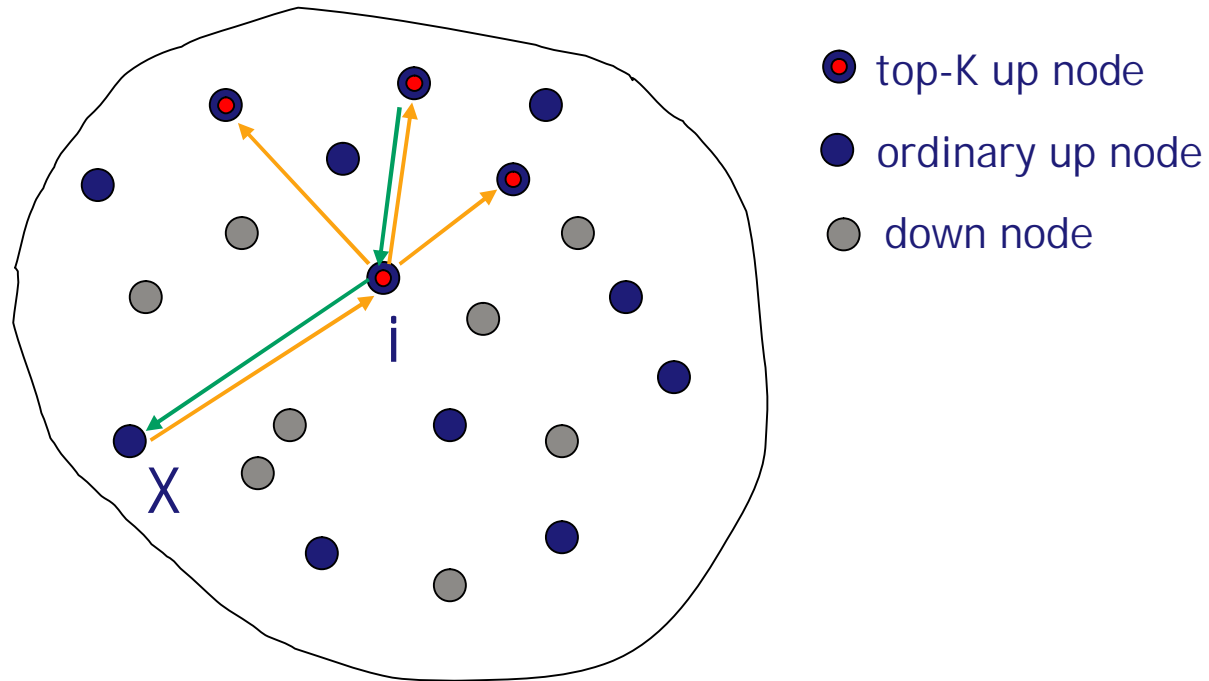
Object o tends to get replicated in its attractor nodes.

Queries for o tend to be sent to attractor nodes.
è tend to get hits

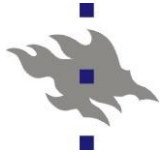
Problem: Can miss even though object is in an up node in the community



Top-K Algorithm

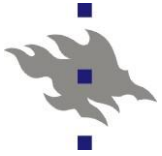


- If i doesn't have o , i pings top-K winners.
- i retrieves o from one of the top-K if present.
- If none of the top-K has o , i retrieves o from outside.

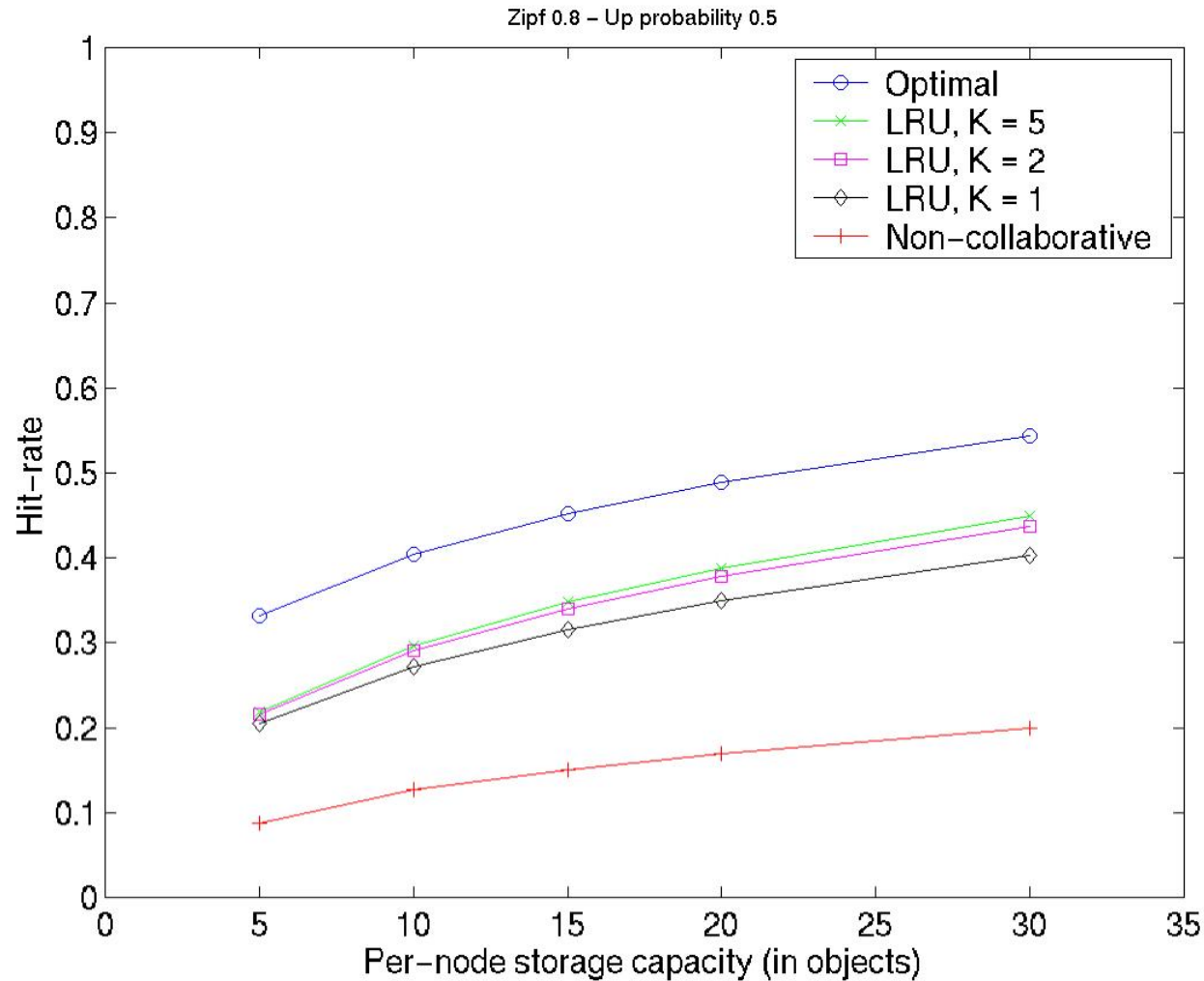


Simulation

- n Adaptive and optimal algorithms
- n 100 nodes, 10,000 objects
- n Zipf = 0.8, 1.2
- n Storage capacity 5-30 objects/node
- n All objects the same size
- n Up probs 0.2, 0.5, and 0.9
- n Top K with $K = \{1, 2, 5\}$



Hit-Probability vs. Node Storage

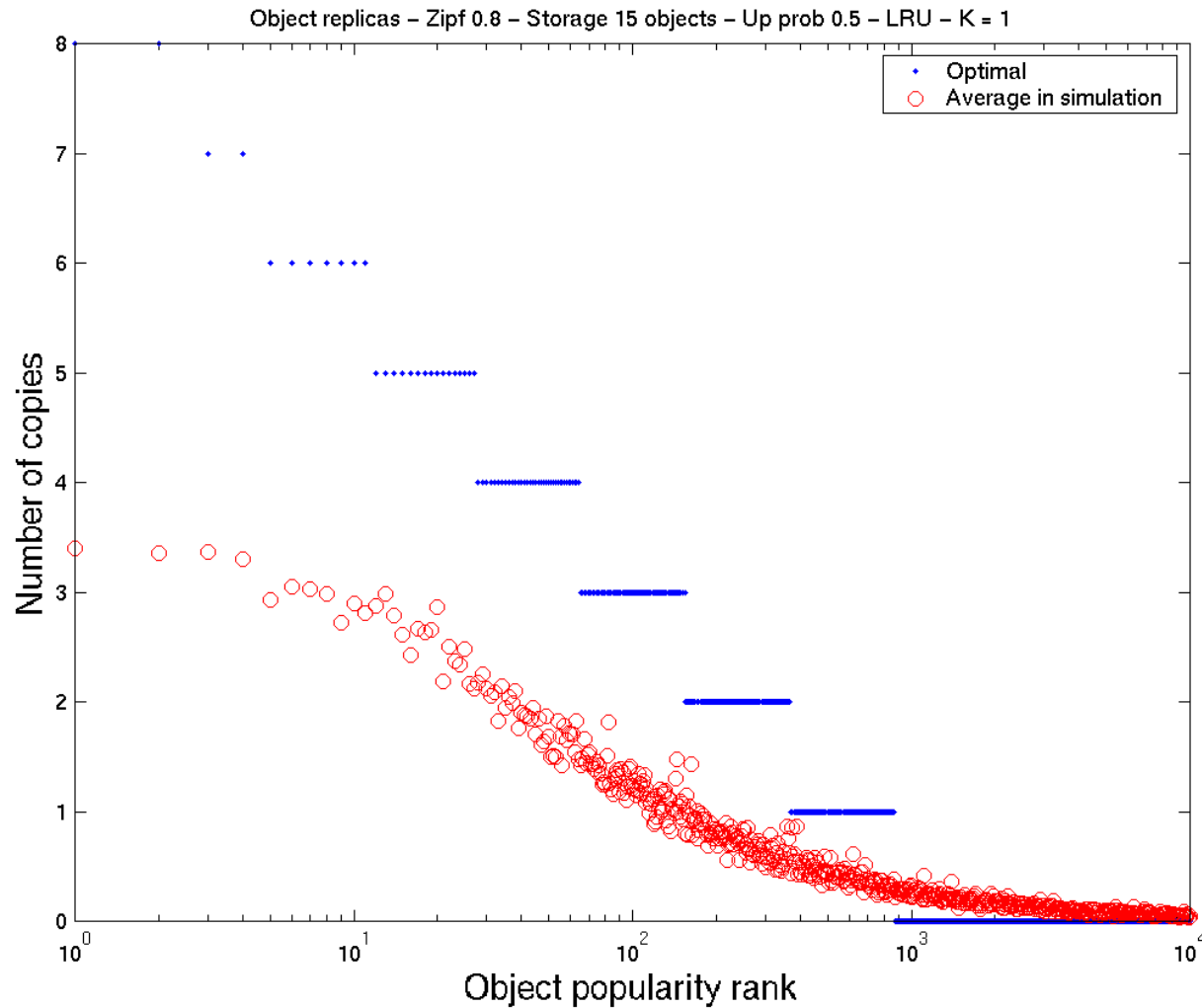


$$p = P(\text{up}) = .5$$

$$\text{Zipf} = .8$$



Number of Replicas

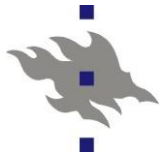


$$p = P(\text{up}) = .5$$

15 objects per node

$$K = 1$$

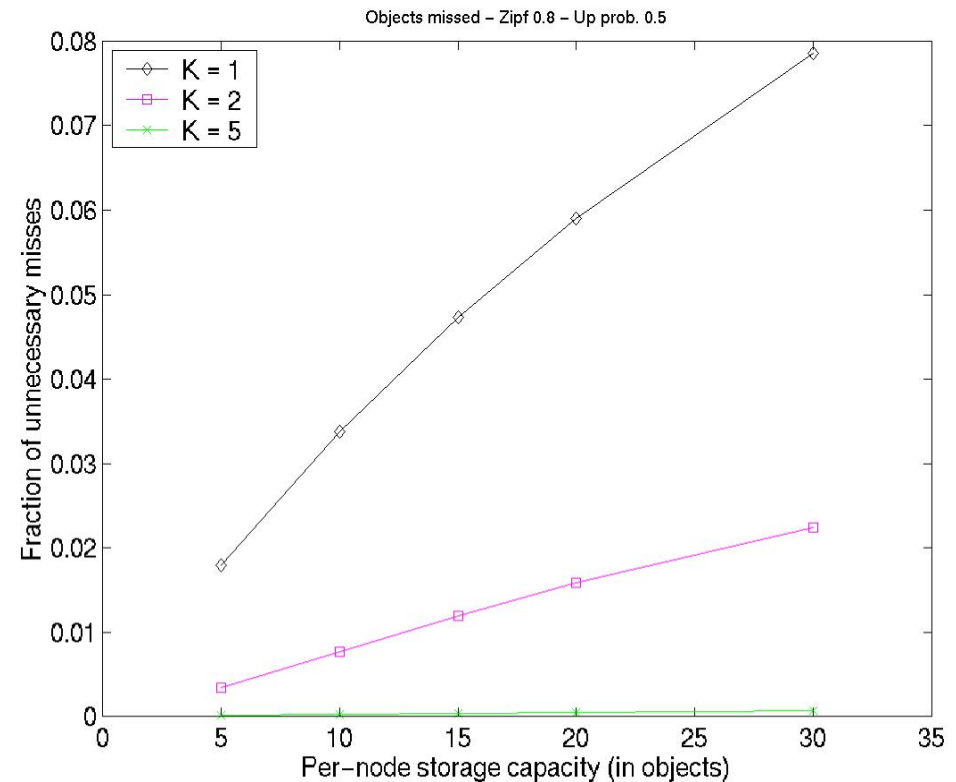
$$\text{Zipf} = .8$$

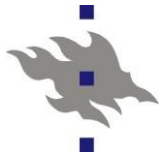


General observations

- Community improves performance significantly
- LRU lets unpopular objects linger in peers
- Top-K algorithm is needed to find object in aggregate storage (see right)

How can we do better?





Most Frequently Requested (MFR)

- Each peer estimates local request rate for each object

 - denote $\lambda_o(i)$ for rate at peer i for object o

- Peer only stores the most requested objects

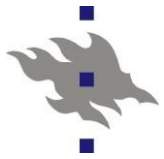
 - packs as many objects as possible

Suppose i receives a request for o :

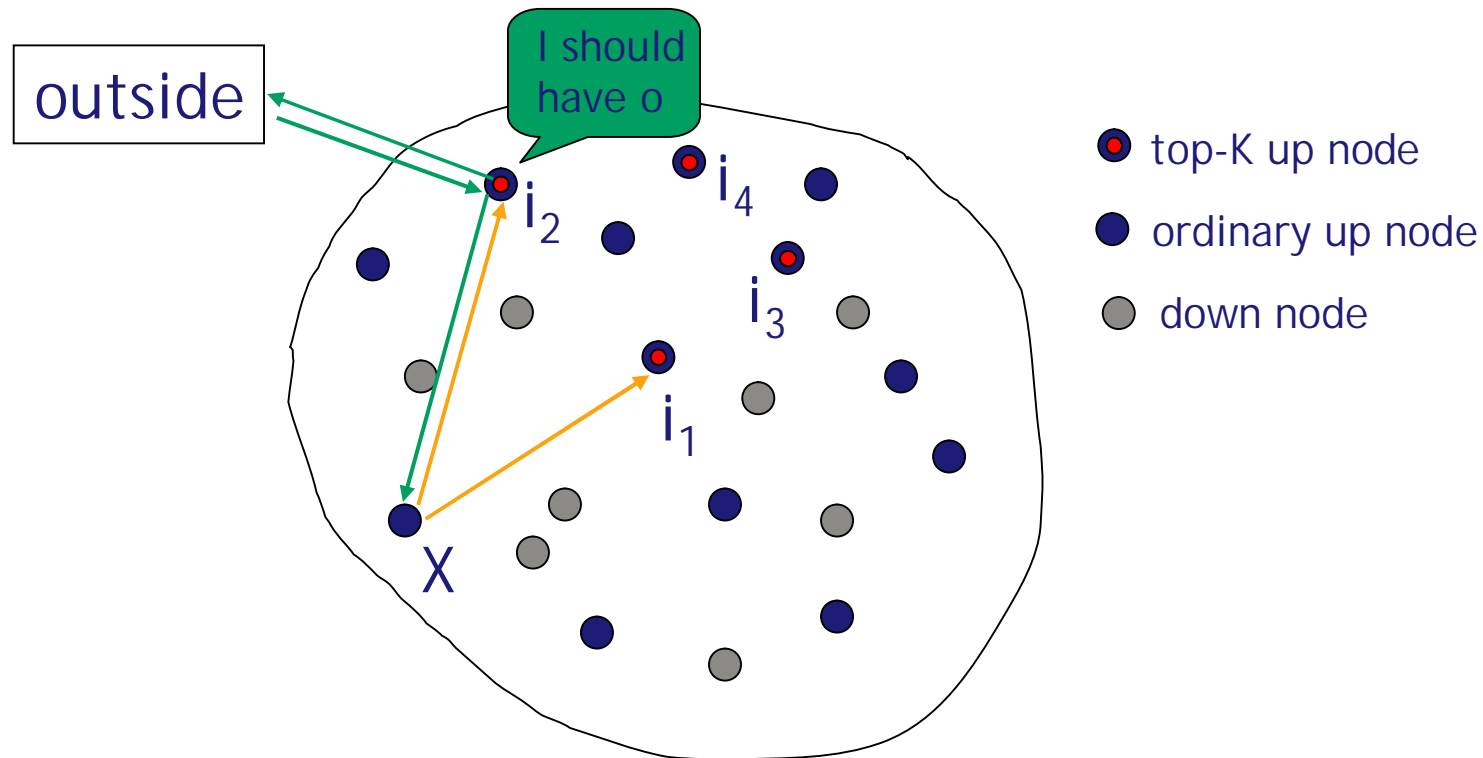
- i updates $\lambda_o(i)$

- If i doesn't have o & MFR says it should:

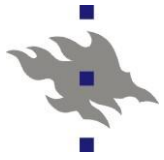
 - i retrieves o from the outside



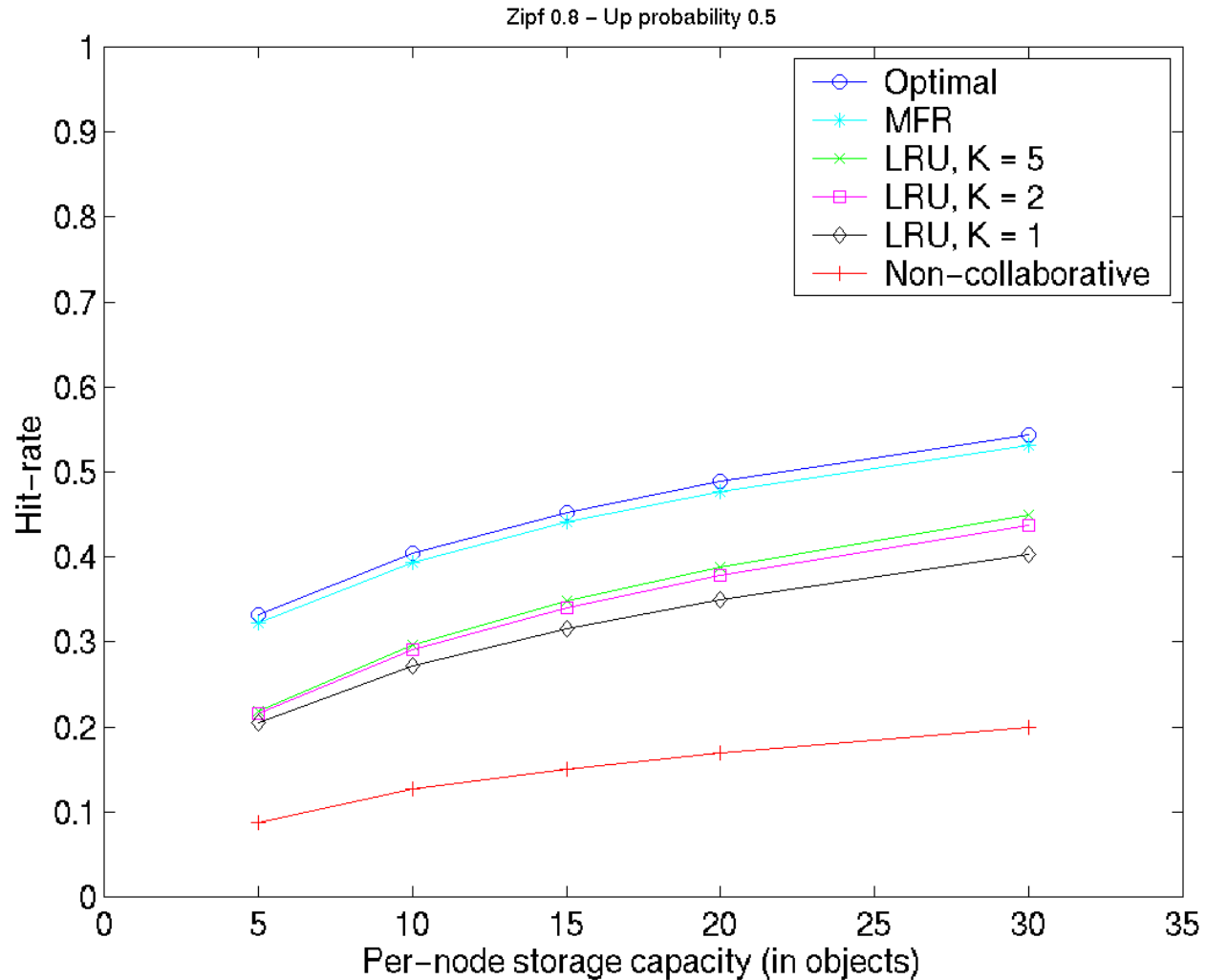
Most-Frequently-Requested Top-K Algorithm



MFR combines replacement and admission policies



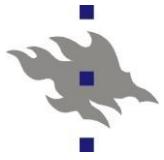
Hit-Probability vs. Node Storage



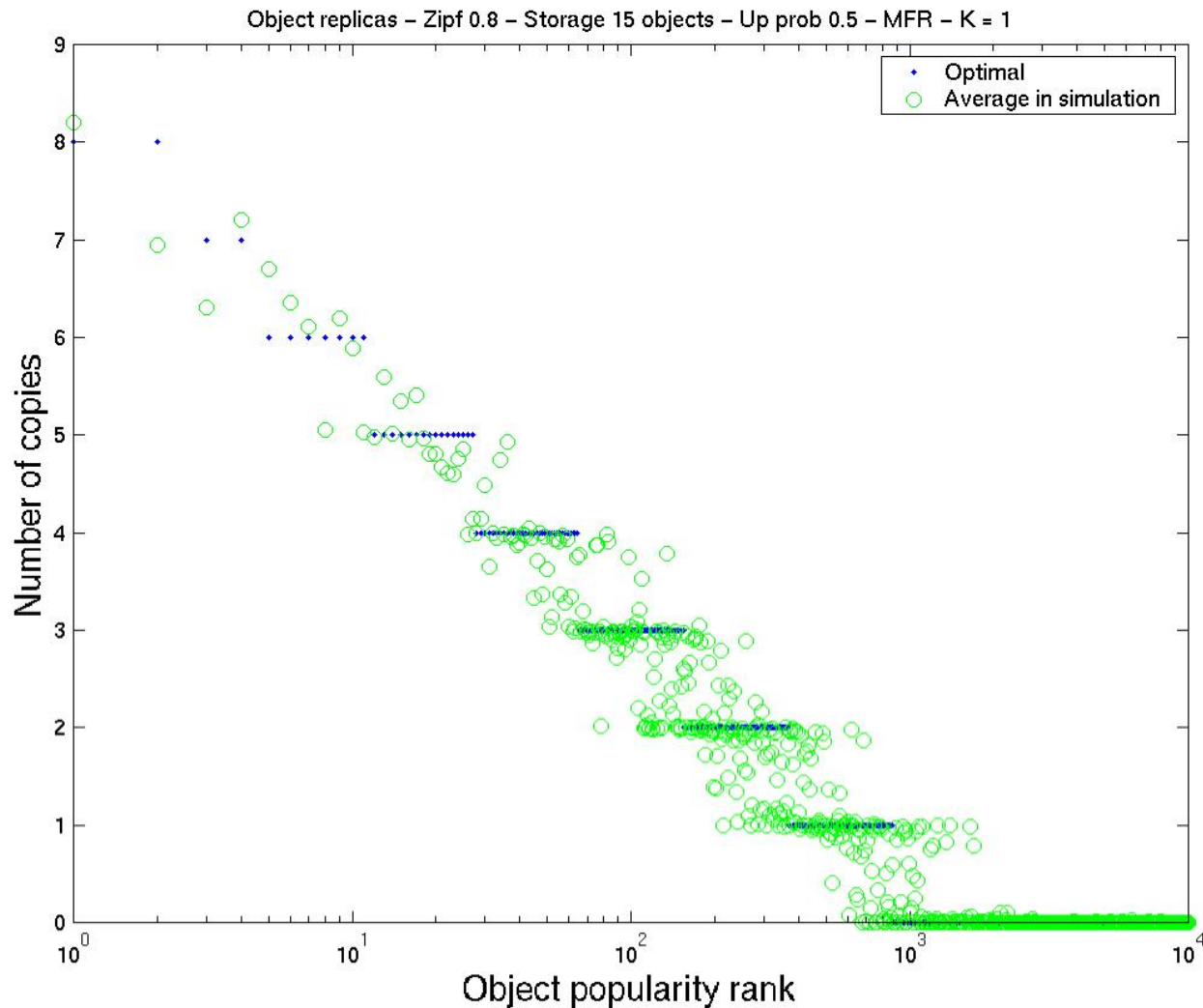
$p = P(\text{up})$
= .5

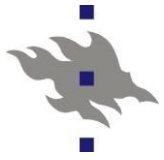
MFR: $K=1$

Zipf = .8



Replica Profile





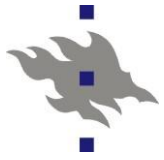
Summary: MFR Top-K Algorithm

Implementation

- n Layers on top of location substrate
- n Decentralized
- n Simple: each peer keeps track of a local MFR table

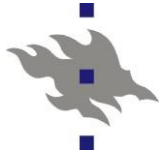
Performance

- n Provides near-optimal replica profile



Optimality of MFR

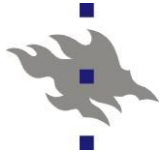
- n Recall basic idea of MFR:
 - n Each peer estimates local request rate for each object
- n Analytical procedure for MFR Top- l : (all nodes)
 - n Init: $\lambda_j = q_j/b_j$, $j = 1, \dots, J$, and $T_i = S_i$, $i = 1, \dots, I$
 1. Find file j with largest λ_j
 2. Sequentially examine winners for j until $T_i \geq b_j$ and $x_{ij} = 0$
 - Set $x_{ij} = 1$
 - Set $\lambda_j = \lambda_j(1-p_i)$
 - Set $T_i = T_i - b_j$
 - If no such node, remove file j from consideration
 3. If still files to be considered go to step 1, otherwise stop.



Evaluation

- n Suppose all files are same size
- n Suppose no ties in step 1 (λ_j)
- n Then Top- k MFR converges to previous procedure
 - à Faster way to evaluate performance

- n Comparing Top- k MFR to true optimal solution:
 - n Almost always gives optimal result (95%)
 - n Simple counter-example: Top- k MFR \neq optimal



Top- k MFR and Non-Optimality

- Assume 2 nodes and 4 objects
 - Each node can store 2 objects, both up prob. 0.5
- Assume request probabilities and winners as shown:

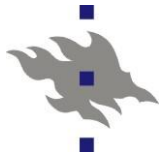
Object	Req. Prob.	1st Winner
1	5/13	1
2	3/13	2
3	3/13	2
4	2/13	1

- What does Top- k MFR do and what is optimal?



Solution

- n Top-/ MFR places objects in the order of popularity
 - n Object 1 --> Node 1, Object 2 --> Node 2, Object 3 --> Node 2
 - n Next would be Object 1 again (reduced request rate $5/13 * 1/2$)
 - n But only node 1 has space and there is already copy of 1 there
 - n Hence, Top-/ MFR puts Object 4 --> Node 1
- n Optimal solution is:
 - n Object 1 --> Node 1 and 2, Object 2 --> Node 1, Object 3 --> Node 2
- n As mentioned above, similar cases appear even in bigger communities
 - n But problem typically “1 copy too much for object X and 1 copy too little for object Y”

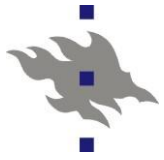


Continuous Optimal

Let $y_j = b_j n_j$, and treat y_j as continuous variable.

$$\begin{array}{ll} \text{Minimize} & \sum_{j=1}^J f_j(y_j) \\ \text{subject to} & \sum_{j=1}^J y_j = S \quad y_j \geq 0, j = 1, \dots, J \end{array}$$

where $f_j(y_j) = q_j(1-p)^{y_j/b_j}$



Continuous Optimal (2)

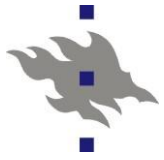
- (1) Order objects according to q_j/b_j
- (2) There is an L such that $n_j^* = 0$ for all $j > L$.
- (3) For $j \leq L$, “logarithmic replication rule”:

$$n_j^* = \frac{s}{B_L} + \frac{\sum_{l=1}^L b_l \ln(q_l / b_l)}{B_L \ln(1-p)} + \frac{\ln(q_j / b_j)}{\ln(1/(1-p))}$$

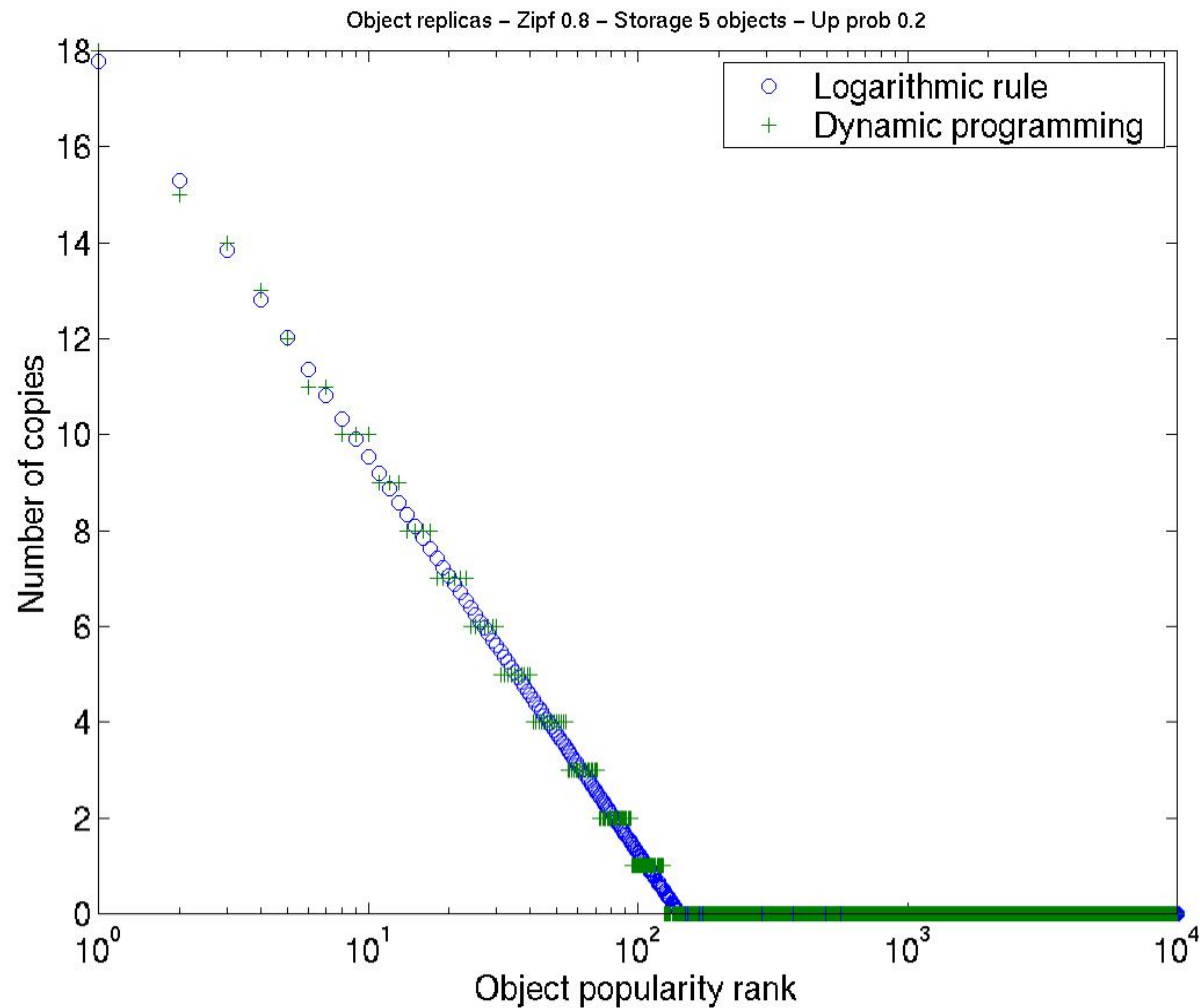
$$= K_1 + K_2 \ln(q_j / b_j)$$



Logarithmic replication rule



Continuous Optimal vs. Discrete



Continuous gives upper bound

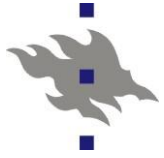
Bound usually tight

Differences are due to discrete being constrained to integer values



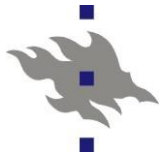
Replication Summary

- n Adaptive replication in communities
 - n Peers in community download content
 - n Content always available in “outside repository”
- n Model of optimal replication of content
 - n Which peers should hold which objects
 - n Model as an integer programming problem (NP-complete)
- n Approximation with “homogeneous case”
 - n Optimal solution with dynamic programming
- n Several different algorithms for comparison
 - n Simple LRU
 - n Top-K LRU
 - n MFR (best performance)



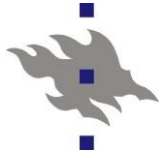
Replication: General Comments

- n Studied two cases:
 - n Static replication, all files equally important
 - n Dynamic, on-the-fly replication, some files more popular
- n Different goals in the two cases
 - n Highest possible availability, no storage constraints
 - n Provide high hit-rate, only limited storage
- n For first case, adding a storage constraint would limit number of files that can be stored
 - n All the rest of the analysis and results remain unaffected
- n What can we learn?



Replication: Lessons

- n When peers mostly up, we need about 5-10 copies
 - n Applies in both cases
 - n Implication: P2P storage system with N GB of capacity can store about N/5 or N/10 GB of data
 - n Maintenance cost of reliable file server vs. extra hard disk?
- n When peers mostly down, we need \gg 100 copies for high availability
 - n This is more realistic for global P2P network (in today's world)
 - n For example, if you donate 100 GB to network, you can store:
 - 100 000 emails, OR
 - 1000 digital photos, OR
 - 300 MP3 files, OR
 - 1 movie (DivX) files (or \sim 0.25 movie in DVD quality)
 - n Not efficient at all...



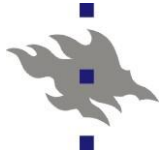
Replication: Future

- n What are the implications for P2P storage systems?
- n “No problem” in corporate environments
 - n Lot of computers with good resources and high uptime
 - n Cost of reliable file servers very high
 - n P2P storage comes “for free”
- n Wide area storage?
 - n Most of analysis assumes no additional coordination
 - n Storage invariants can reduce number of copies
 - n Must have additional coordination to make system attractive
 - n Is factor-of-10 reduction in capacity acceptable to users?
 - Most home users don't care about reliability, don't take backups
 - Most home users wouldn't see benefits?



Load Balancing

- n What if the first place winner for a popular object is (almost) always up?
- n Problem: How to balance the load between the peers in the community?
- n In fact, what is the goal of a load balancing algorithm?
 1. Make everyone do the same amount of work?
 - n But: Peers might be heterogeneous
 2. Allow individual peers to determine their own load?
 - n Problem: Too much refused traffic hurts performance
- n **Two approaches:**
 - n Fragmentation
 - n Overflow



Load Balancing: Solutions

n Fragmentation

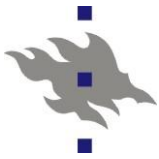
- n Idea: Divide each object into chunks, store chunks individually
- n One chunk is much smaller than a file, hence load is balanced better, since chunks are stored on different peers
- n Achieves overall load balancing (goal 1 from above)

n Overflow

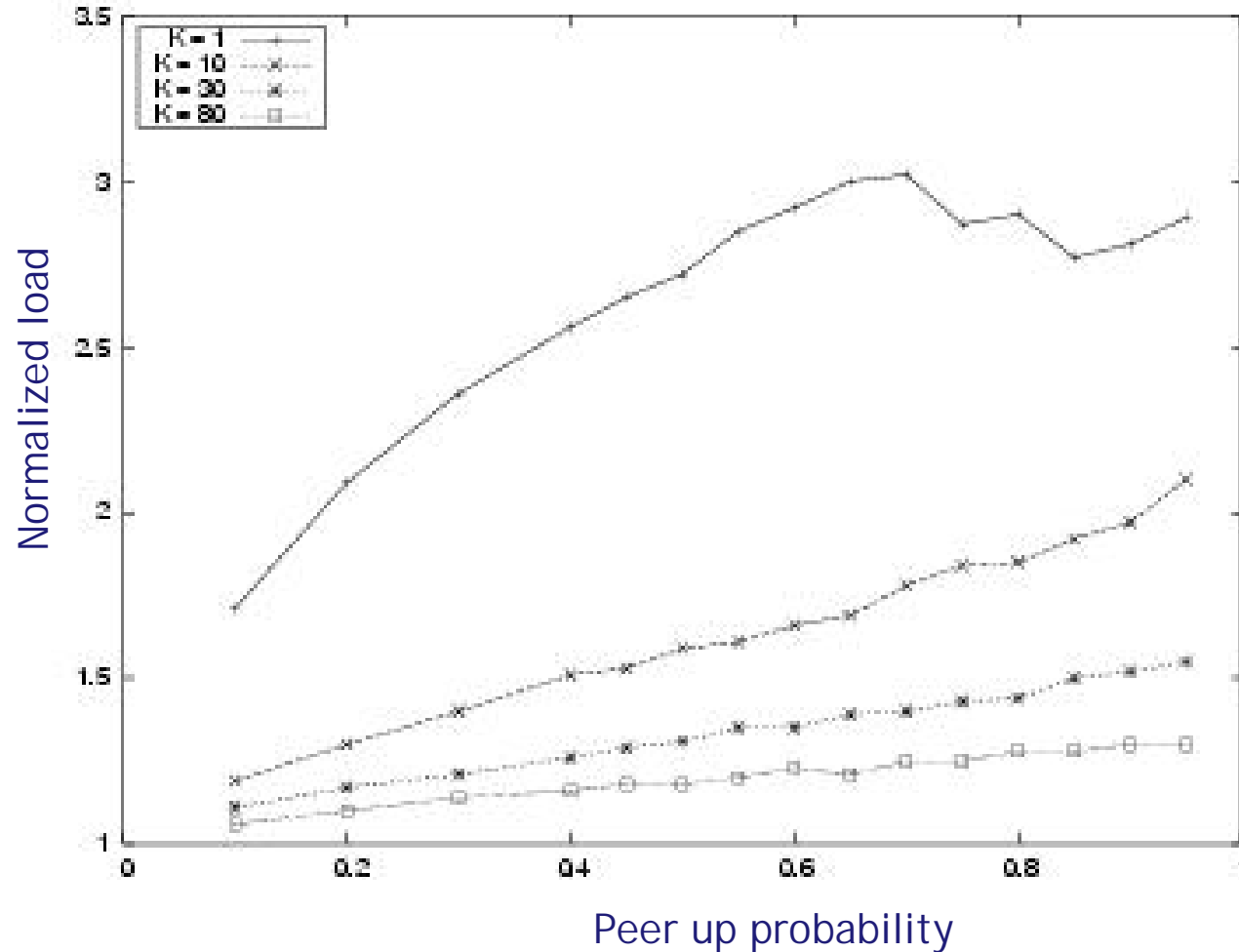
- n Idea: Allow peers to refuse requests
- n Request passed on to the next winner (eventually to outside)
- n Allows a peer to decide how much traffic to handle
- n Achieves goal number 2 from above

n Fragmentation + Overflow

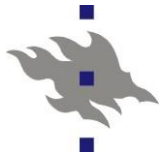
- n Use both approaches



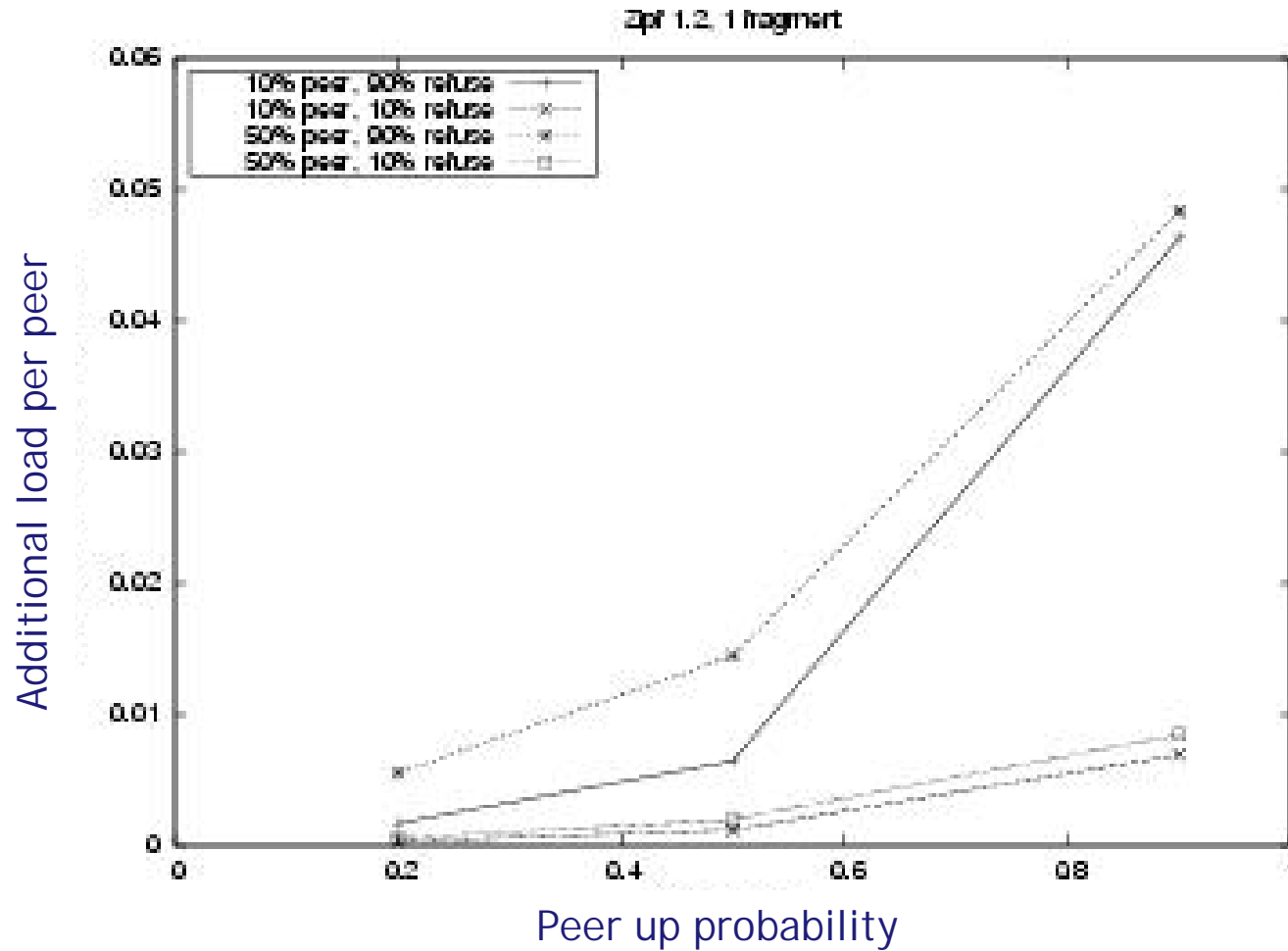
Load Balancing: Fragmentation



- n 90-percentile load for Zipf parameter 1.2
- n K = number of chunks
- n Load normalized to “fair share”
- n Seems to work quite well for large number of chunks
- n Large files --> many chunks



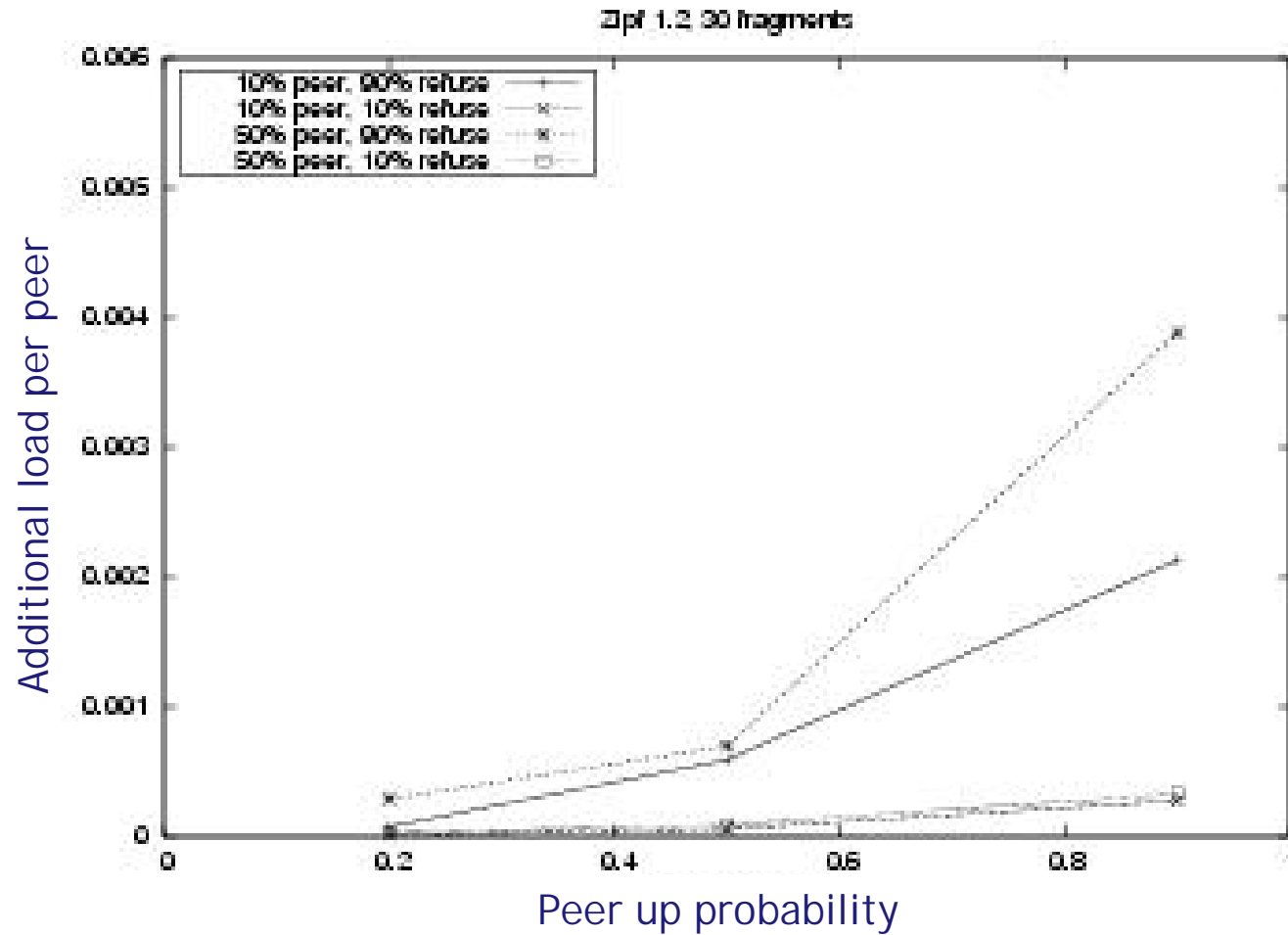
Load Balancing: Overflow



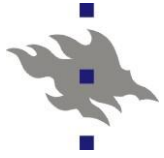
- Overflow with 1 chunk
- Different amounts of refused traffic
- Worst case: 5% additional load for each peer



Fragmentation + Overflow

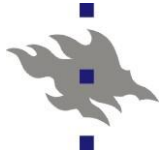


- n Same as above, but with 30 chunks per file
- n Additional load less than 0.5% in all cases



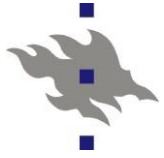
Overflow: Refused Traffic

- n When large number of traffic is refused, it goes to the outside, thus reducing hit-rate
- n How much is hit-rate affected?
- n Rough rule of thumb: Proportion of reduced traffic reduces overall storage capacity by the same proportion
- n Example: If 50% of peers are refusing 50% of the traffic, then overall storage capacity is reduced by 25%



Load Balancing: Summary

- n Without any load balancing mechanism, load is severely unbalanced
- n Fragmentation approach works well for achieving a uniform load on all peers
- n Pure overflow approach allows individual peers to reduce their load at a cost of increased load to others
- n Overflow with fragmentation works best
- n Refused traffic ends up effectively reducing the overall amount of storage offered by the community



Chapter Summary

- n Performance evaluation of P2P systems
- n DHT performance under heavy load
 - n Evaluate effects of different parameters
 - n Evaluate DHT-based applications
- n Storage systems
 - n Unconstrained system
 - Provide target availability
 - n Constrained system, P2P community
 - Maximize hit-rate
 - n Load balancing