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

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

DHT Performance Issues

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

DHT Performance

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

Cost vs. Performance

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

Comparison

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

Tapestry

- Base, stabilization interval, backup nodes

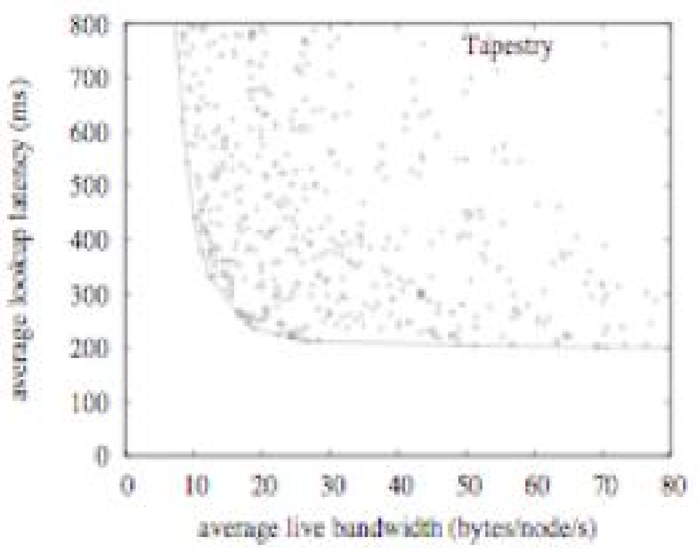
Chord

- Number of successors, finger base

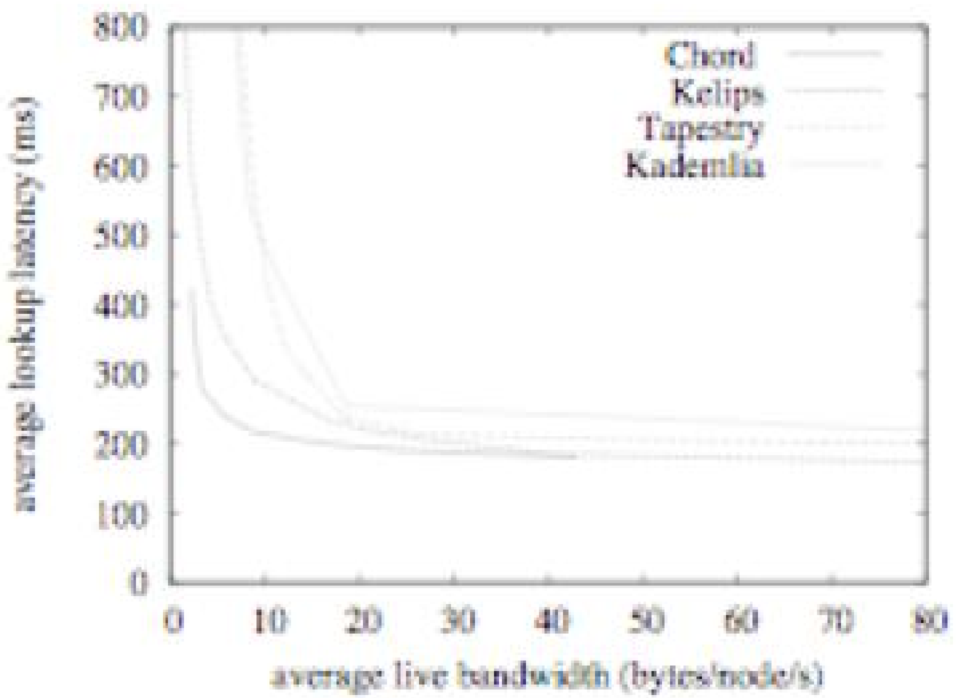
Evaluation Parameters

- 1024 nodes in network
- Only key lookup, no data retrieval
- Nodes request random keys
  - Exponentially, mean 10 mins
- Nodes join and leave
  - Exponentially, mean 1 hour
- 6 hours simulated time
- Nodes keep IP and ID

- Many parameter combinations
- No single best choice
- Many optimal choices
- Best points on *convex hull* of all points



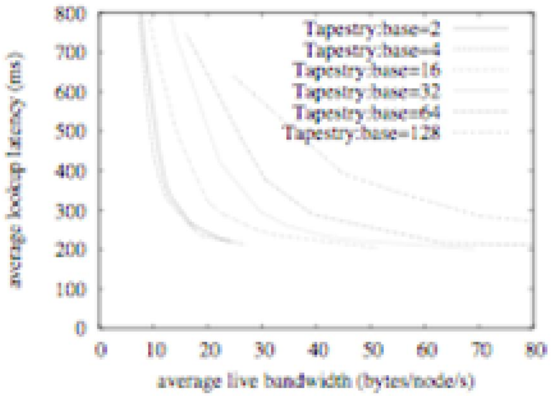
Overall Results



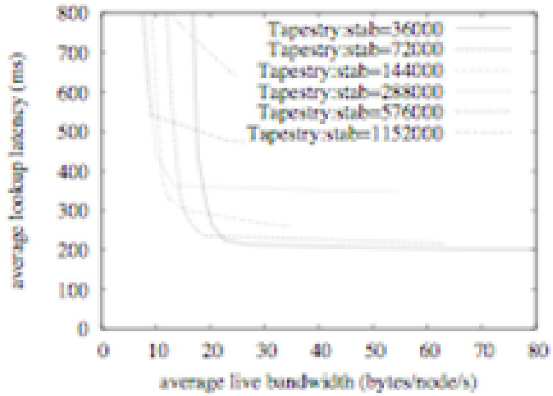
- All 4 DHTs shown
- Chord is “best”
  - Kademlia uses iterative routing, recursive appears to be better
- Any DHT can be below 250ms latency
  - Some need lot of bandwidth
- Chord uses bandwidth efficiently
  - Finger tables not needed
  - Successor pointers maintain correctness, low bandwidth required
- 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



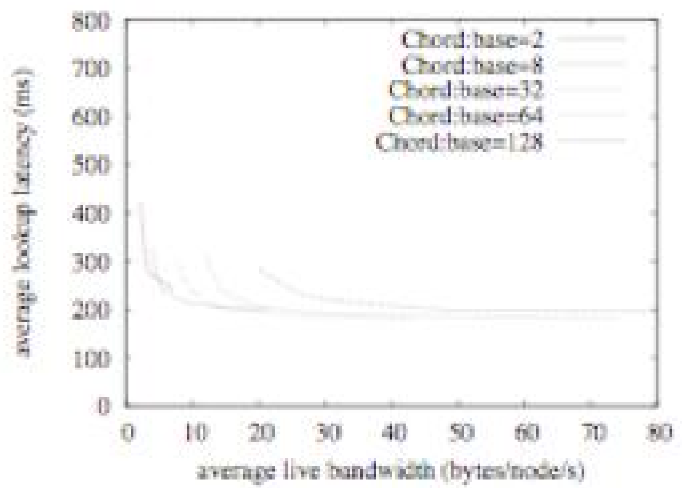
Base



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

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

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

Performance of DHT-Based Applications

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

P2P DNS

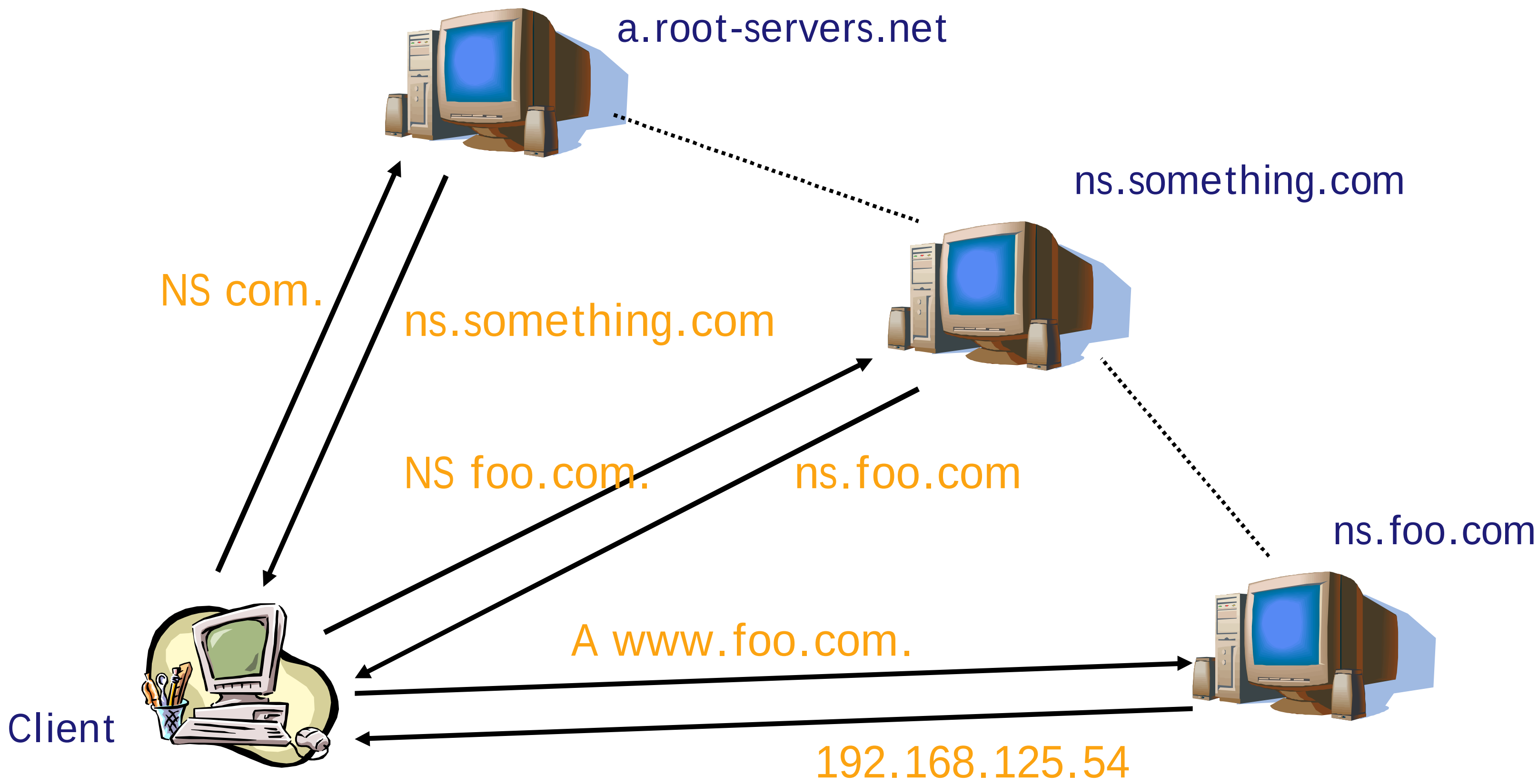
- Domain Name System (DNS) very much client-server
- *Ownership of domain = responsibility to serve its data*
- DNS concentrates traffic on root servers
  - Up to 18% of DNS traffic goes to root servers
- Lot of traffic also due to misconfigurations

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

DNS: Overview

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

DNS: Example



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

How to Do P2P DNS?

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

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

P2P DNS: Performance

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

Why (Not) P2P DNS?

Pros

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

Cons

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

Future of DHT-Based Applications?

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

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

Reliability of P2P Storage

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

Questions:

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

Reliability of Data in DHT-Storage

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

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

Assumptions

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

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

- User may need to access several objects to complete one user-level action
  - 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

- We look at only the first three cases
- What are the probabilities of each one of them?

Probabilities of Loss

1. All k peers are down when B reads

$$p_{l1} = (1 - p)^k$$

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

$$p_{l2} \approx \sum_{i=k}^{(1-p)l} \binom{(1-p)l}{i} \left(\frac{p(1-p)}{l} \right)^i$$

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

$$p_{l3} = \sum_{i=k}^{N} \binom{N}{i} \frac{1}{l^i}$$

Numerical Values for Loss

I	10^2	10^3	10^4
$p_{l_3} \approx$	10^{-10}	10^{-15}	10^{-20}

p	$p_{l_1} \approx$
0.99	10^{-10}
0.9	10^{-8}
0.5	0.03
0.3	0.17

$p_{l_2} \approx$ (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}

- First case (green) dominates clearly
  - In above tables, $k = 5$
  - For cases 2 and 3 also applies:

    Search more than k nodes to find object

How to Improve?

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

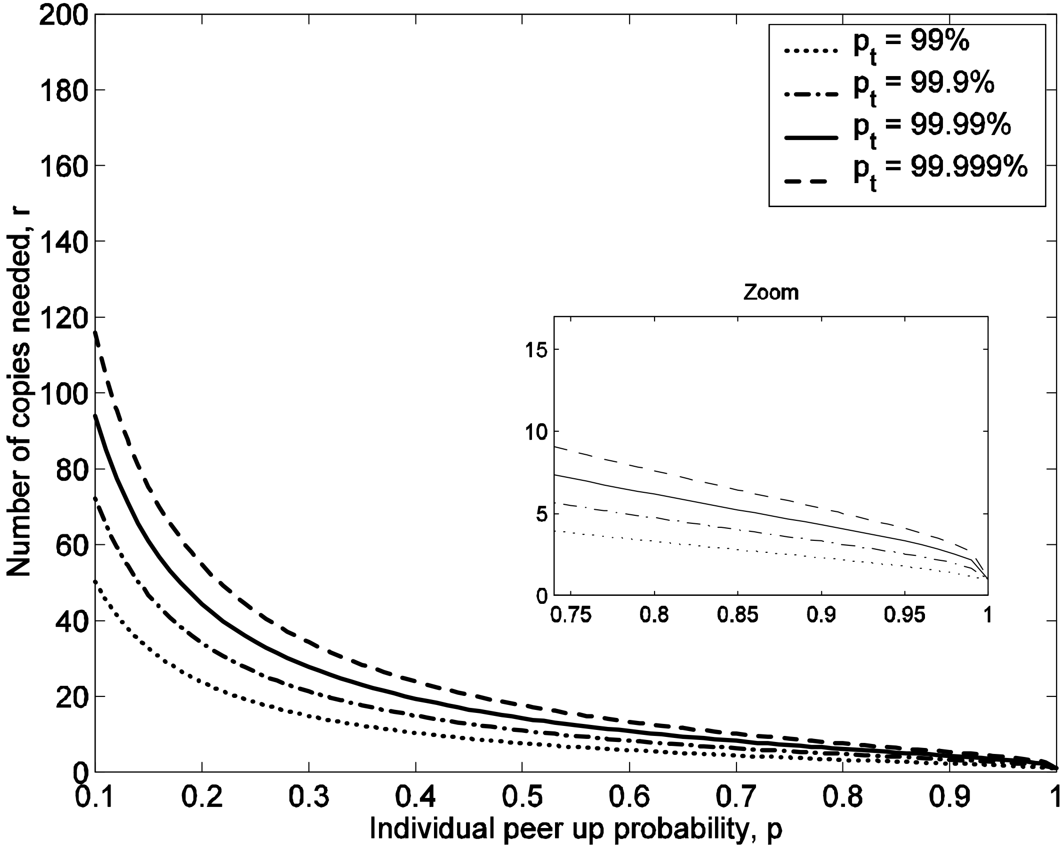
What the User Sees?

- Suppose: User's action needs to access several objects
  - For example, resolve path names of files one level at a time
- For each object: $p_s = 1 - p_{l1} = 1 - (1 - p)^k$
- 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 à small increase in *k*

Replication Summary

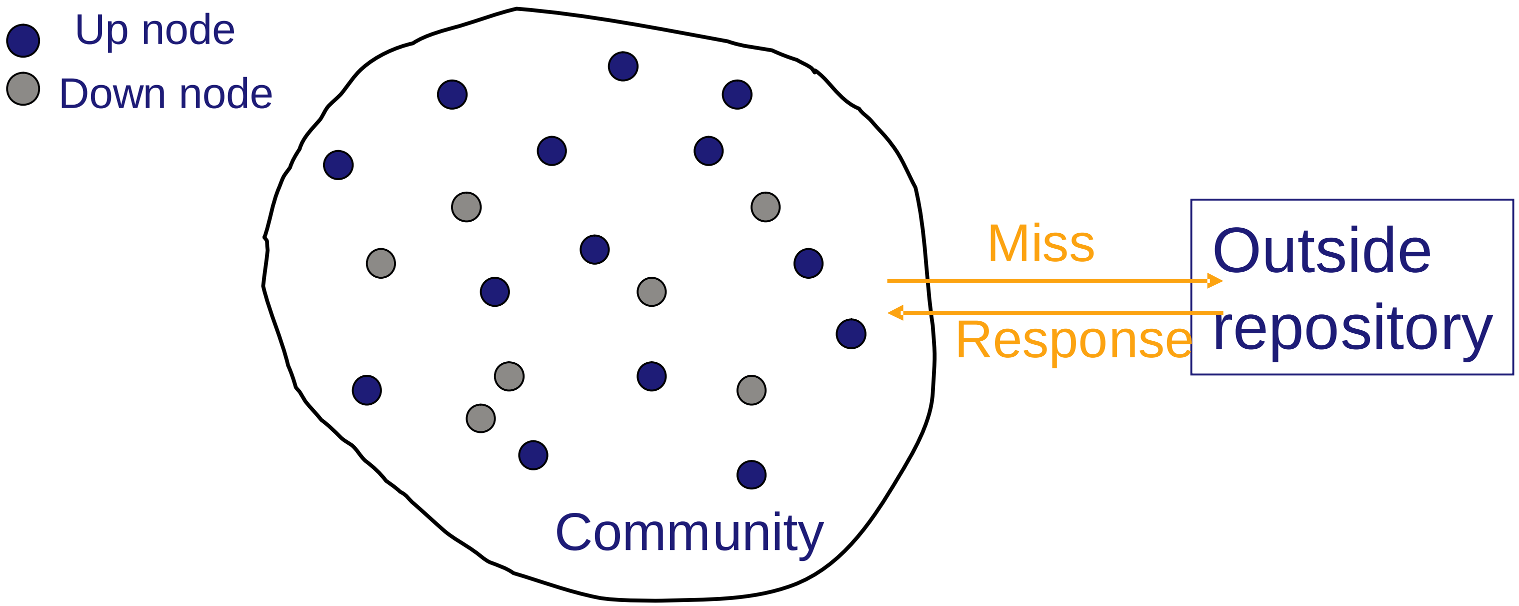
- Replication in read-only system helps availability
- Main cause of unavailability is peers going down
- Create *k* copies of each object
  - If peers mostly up, *k* quite small (< 10)
  - Maintaining actively copies in right peers helps

- Above analysis assumes all objects equally popular or important
  - Not always true
  - Recall: Zipf-distribution for object popularities
  - Also, some objects may require higher availability
- How should objects be replicated in this case?

P2P Content Management

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

Abstract Community Model



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

Replication Issues

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

- What does “optimal replica profile” mean?

Replication Theory

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

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

Optimization Problem

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

subject to $$\sum_{j=1}^{J} b_j x_{ij} \le S_i, \qquad i = 1, \ldots, I$$

$$x_{ij} \in \{0,1\}, \qquad i = 1, \ldots, I, \qquad j = 1, \ldots, J$$

Can be reduced to Integer programming problem: NP

Homogeneous Up Probabilities

- Suppose $p_i = p$
- Let $n_j = \sum_{i=1}^{I} x_{ij}$ = number of replicas of object j
- Let S = total group storage capacity
- Minimize

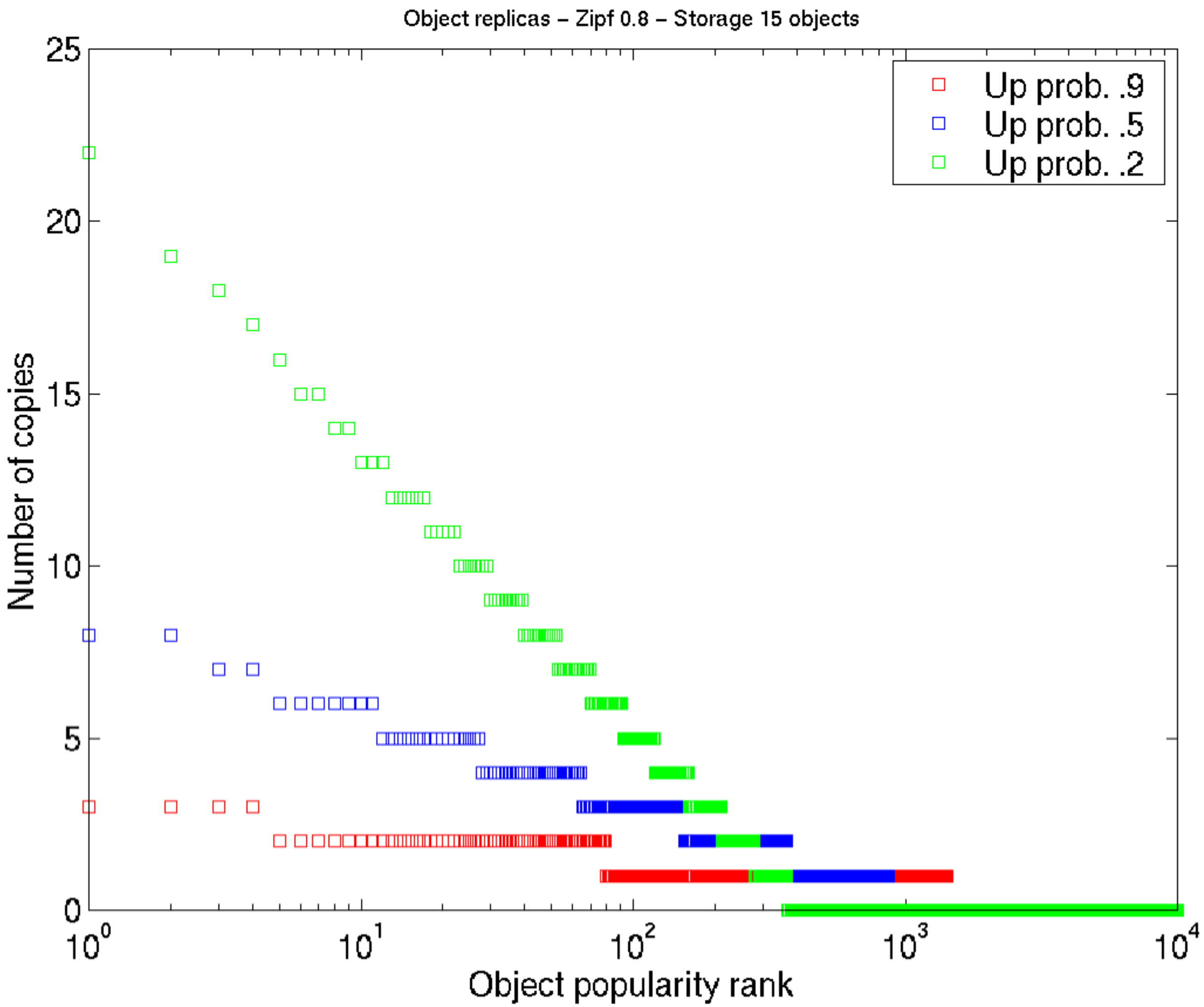
$$\sum_{j=1}^{J} q_j (1-p)^{n_j}$$

Can be solved by dynamic programming

- subject to:

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

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

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

- Need efficient adaptive algorithms!

Assumptions & Goals

Assume

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

Goals

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

DHT: Winners

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

$$\mathrm{hash}(j) \ \text{ð} \ [\ i_j(1),\ i_j(2), \ldots,\ i_j(l)\ ]$$

- 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

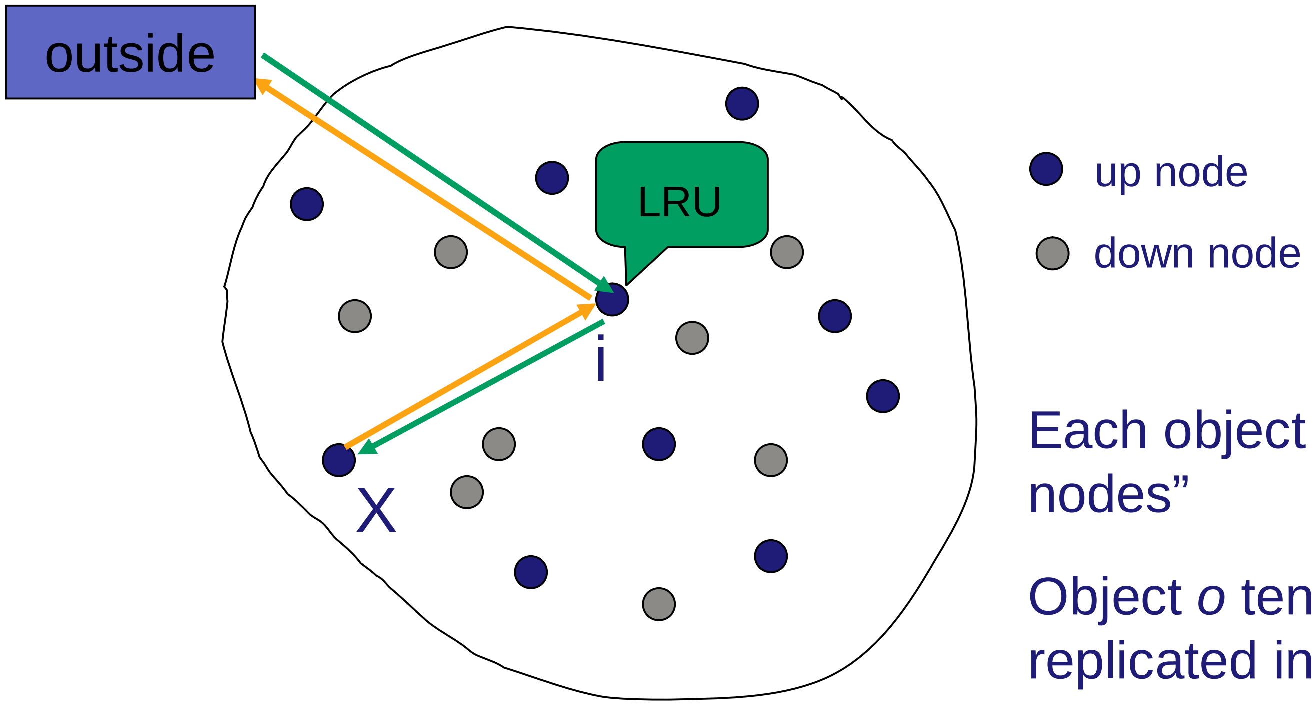
Adaptive Algorithm: Simple Version

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



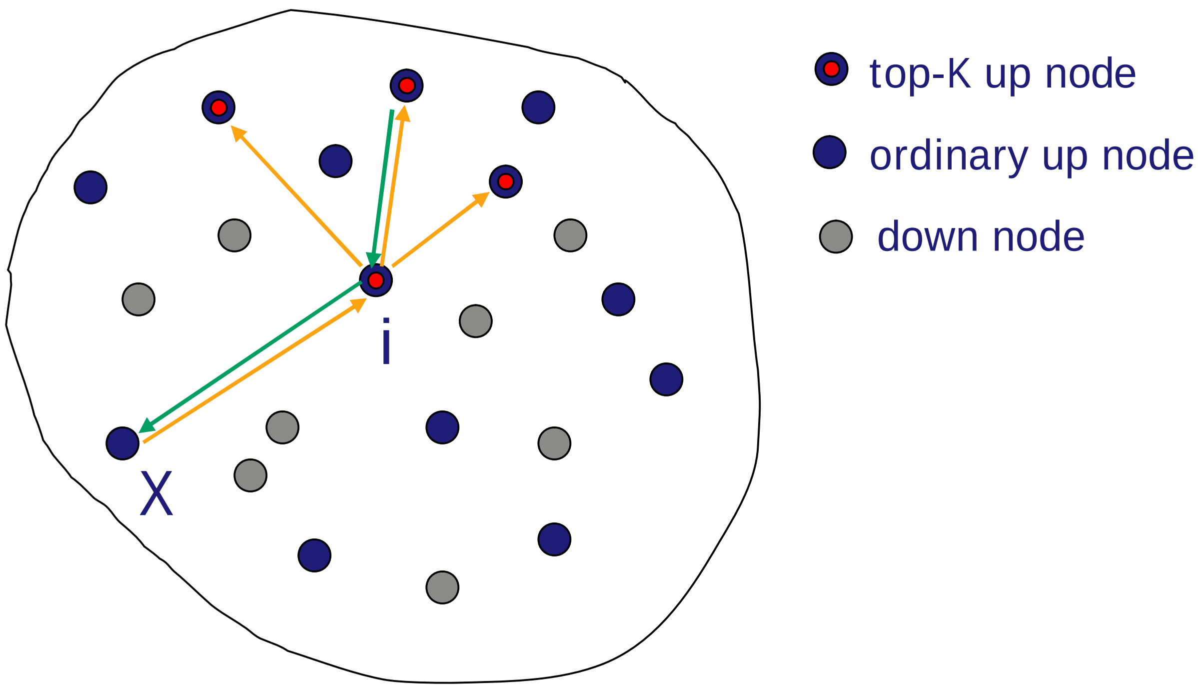
Each object *o* has “attractor nodes”

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

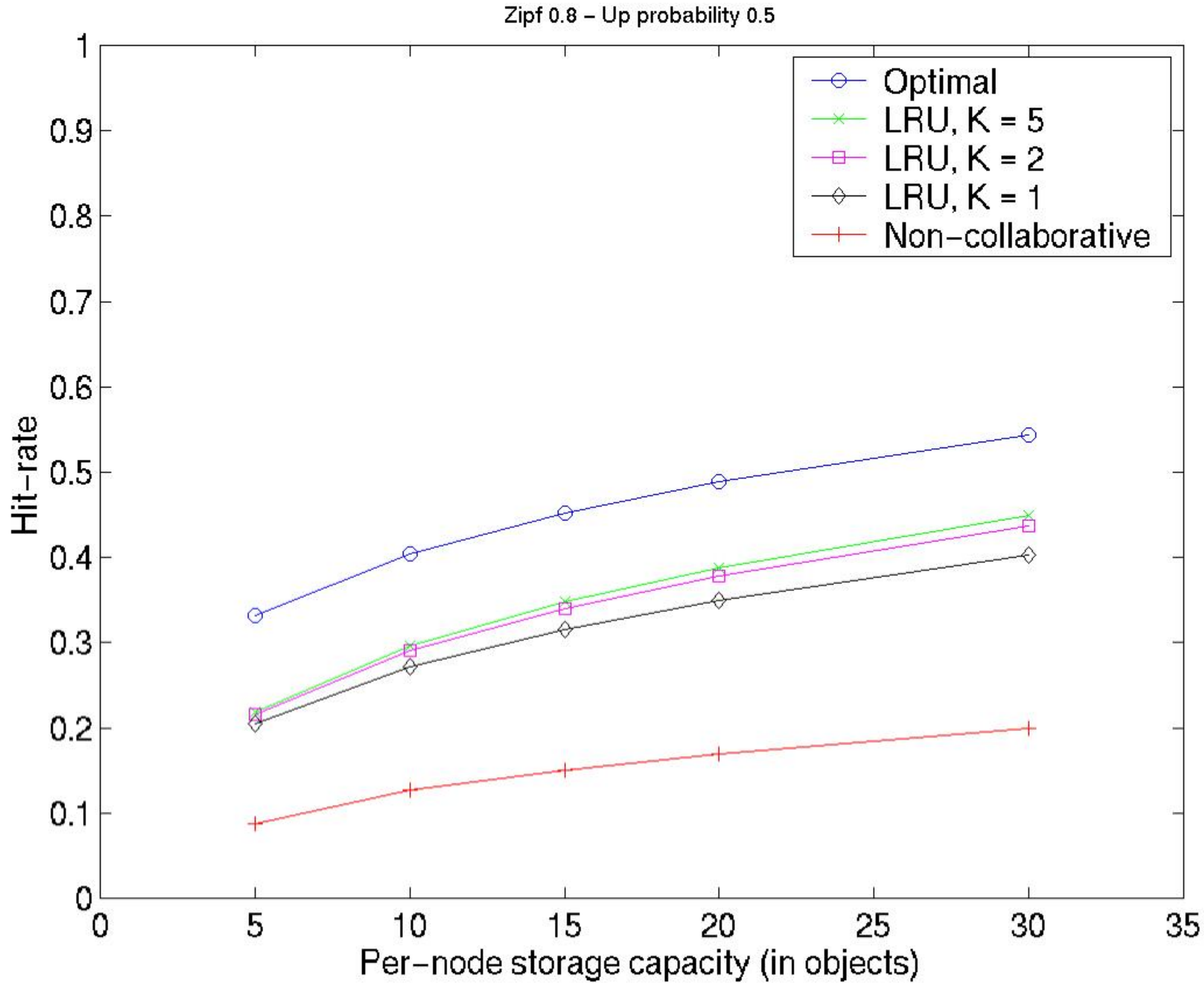


- If *i* doesn't have *o*, *i* pings top-K winners.
- *i* retrieves *o* from one of the top-K if present.
- If none of the top-K has *o*, *i* retrieves *o* from outside.

Simulation

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

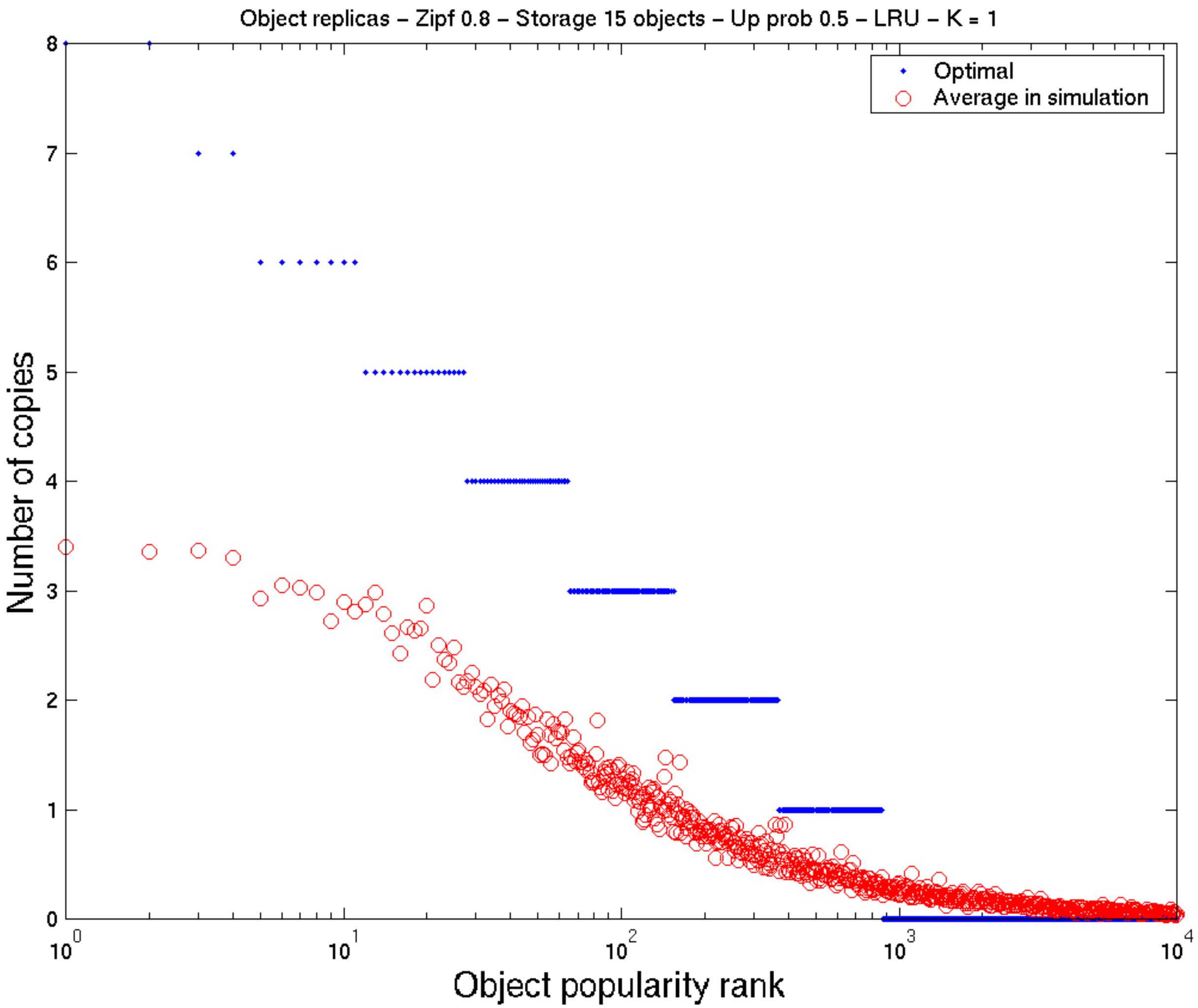
Hit-Probability vs. Node Storage



p = P(up)
= .5

Zipf = .8

Number of Replicas



p = P(up)
= .5

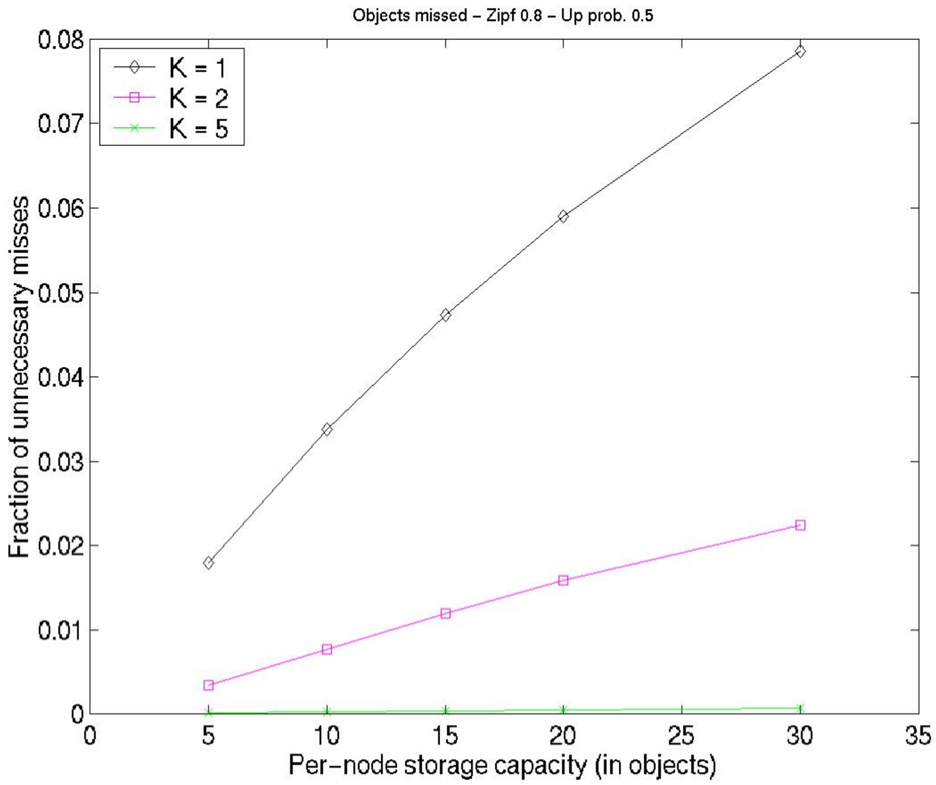
15 objects
per node

K = 1

Zipf = .8

General observations

- Community improves performance significantly
- LRU is 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)

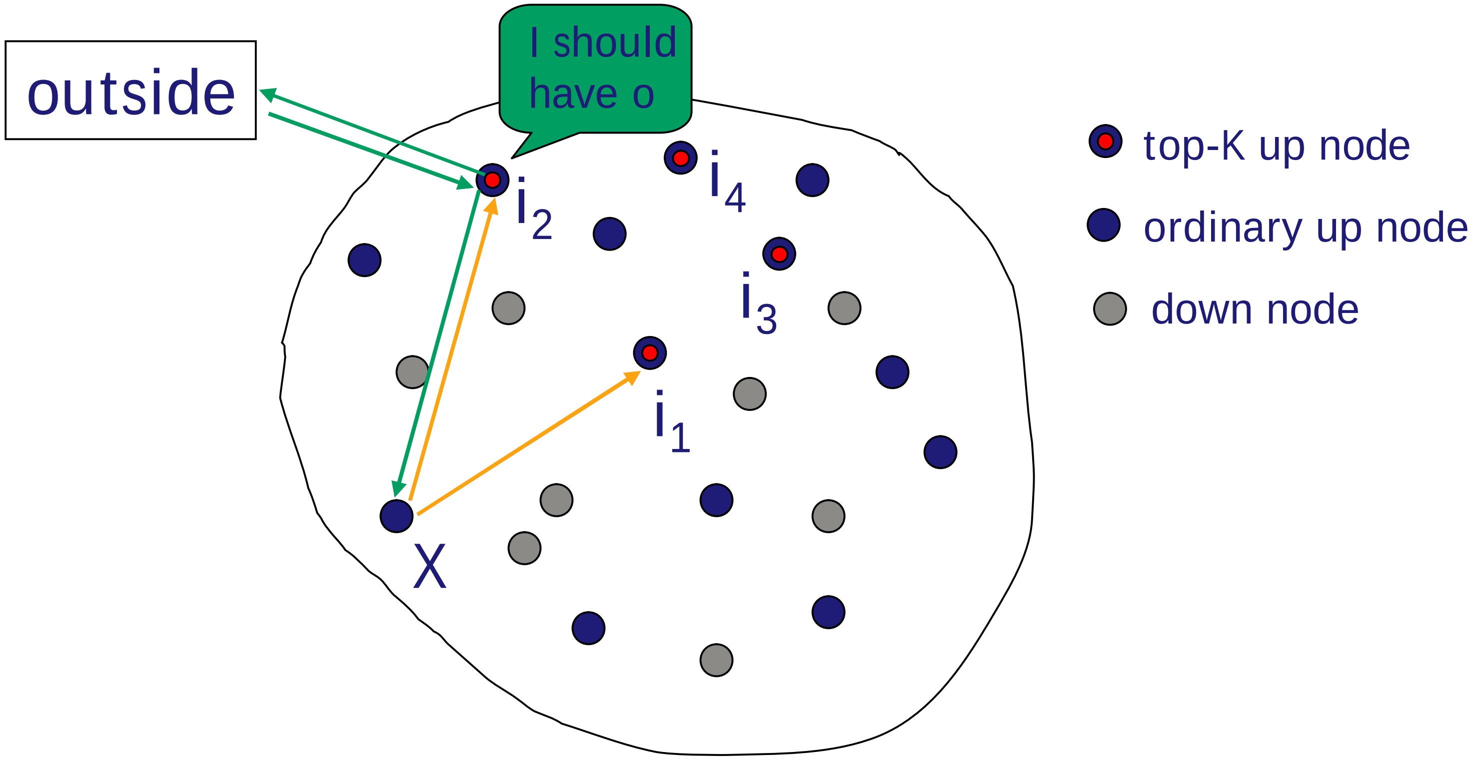
- Each peer estimates local request rate for each object
  - denote $\lambda_o(i)$ for rate at peer *i* for object *o*
- Peer only stores the most requested objects
  - packs as many objects as possible

Suppose *i* receives a request for *o*:

- *i* updates $\lambda_o(i)$
- If *i* doesn't have *o* & MFR says it should:

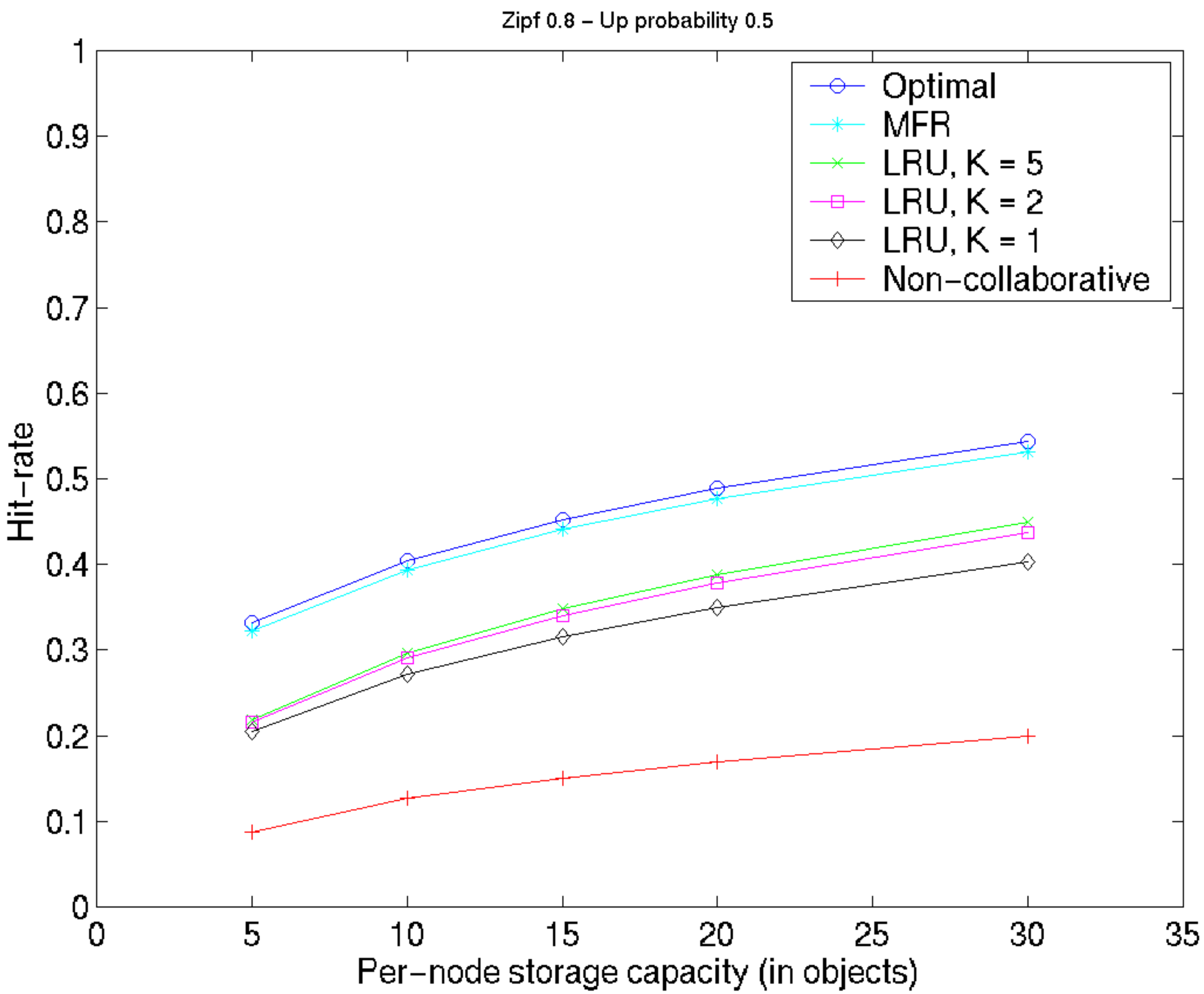
  i retrieves *o* from the outside

Most-Frequently-Requested Top-K Algorithm



MFR combines replacement and admission policies

Hit-Probability vs. Node Storage

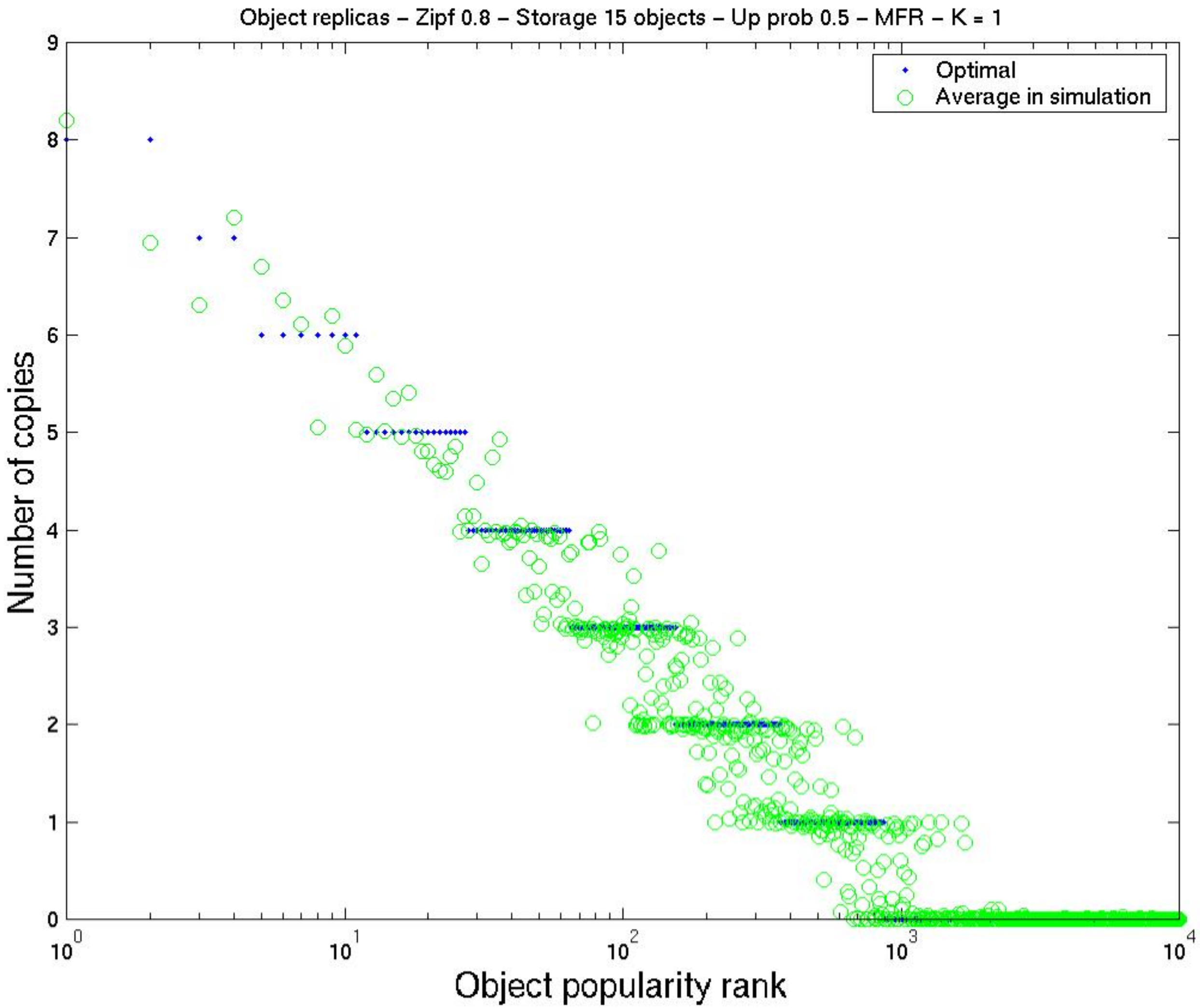


p = P(up)
= .5

MFR: K=1

Zipf = .8

Replica Profile



p = P(up)
= .5

15 objects per node

K = 1

Zipf = .8

Replica profile almost optimal

Summary: MFR Top-K Algorithm

Implementation

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

Performance

- Provides near-optimal replica profile

Optimality of MFR

- Recall basic idea of MFR:
  - Each peer estimates local request rate for each object
- Analytical procedure for MFR Top-I: (all nodes)
  - Init: $\lambda_j = q_j/b_j$, $j = 1, ..., J$, and $T_i = S_i$, $i = 1, ..., I$

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

- Suppose all files are same size
- Suppose no ties in step 1 (λ_j)
- Then Top-*I* MFR converges to previous procedure

  à Faster way to evaluate performance

- Comparing Top-*I* MFR to true optimal solution:
  - Almost always gives optimal result (95%)
  - Simple counter-example: Top-*I* MFR ≠ optimal

Top-*l* 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-*l* MFR do and what is optimal?

Solution

- Top-*l* MFR places objects in the order of popularity
  - Object 1 --> Node 1, Object 2 --> Node 2, Object 3 --> Node 2
  - Next would be Object 1 again (reduced request rate 5/13 * 1/2)
  - But only node 1 has space and there is already copy of 1 there
  - Hence, Top-*l* MFR puts Object 4 --> Node 1
- Optimal solution is:
  - Object 1 --> Node 1 and 2, Object 2 --> Node 1, Object 3 --> Node 2

- As mentioned above, similar cases appear even in bigger communities
  - 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.

Minimize $$\sum_{j=1}^{J} f_j(y_j)$$

subject to $$\sum_{j=1}^{J} y_j = S \qquad y_j \geq 0, j = 1, \ldots, J$$

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

Continuous Optimal (2)

(1) Order objects according to q_j/b_j

(2) There is an L such that $n^*_j = 0$ for all $j > L$.

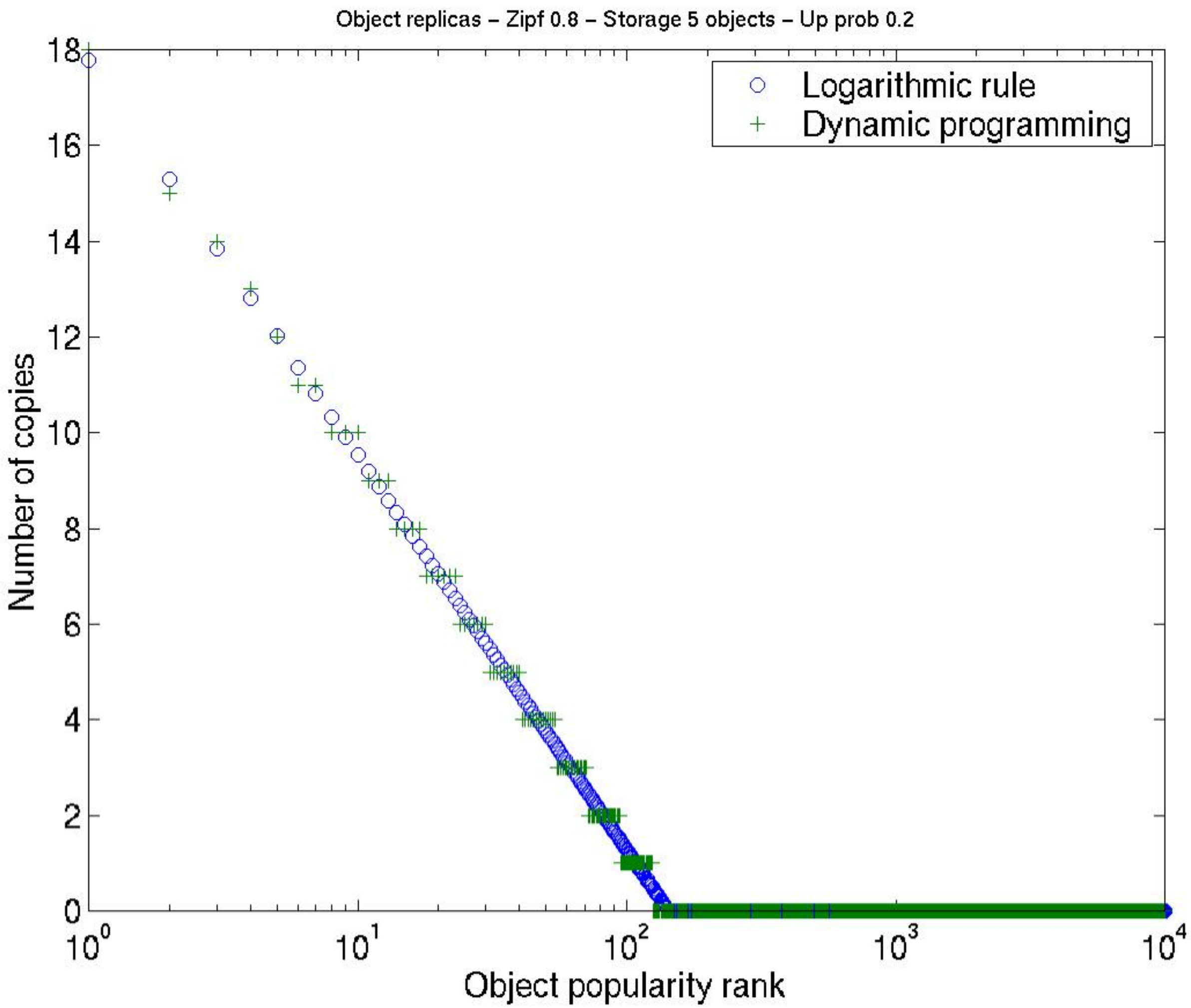
(3) For $j <= L$, "logarithmic replication rule":

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

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

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

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

Replication: General Comments

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

Replication: Lessons

- When peers mostly up, we need about 5-10 copies
  - Applies in both cases
  - Implication: P2P storage system with N GB of capacity can store about N/5 or N/10 GB of data
  - Maintenance cost of reliable file server vs. extra hard disk?
- When peers mostly down, we need >> 100 copies for high availability
  - This is more realistic for global P2P network (in today's world)
  - 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 ~0.25 movie in DVD quality)
  - Not efficient at all…

Replication: Future

- What are the implications for P2P storage systems?
- "No problem" in corporate environments
  - Lot of computers with good resources and high uptime
  - Cost of reliable file servers very high
  - P2P storage comes "for free"
- Wide area storage?
  - Most of analysis assumes no additional coordination
  - Storage invariants can reduce number of copies
  - Must have additional coordination to make system attractive
  - 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

- What if the first place winner for a popular object is (almost) always up?
- Problem: How to balance the load between the peers in the community?
- In fact, what is the goal of a load balancing algorithm?

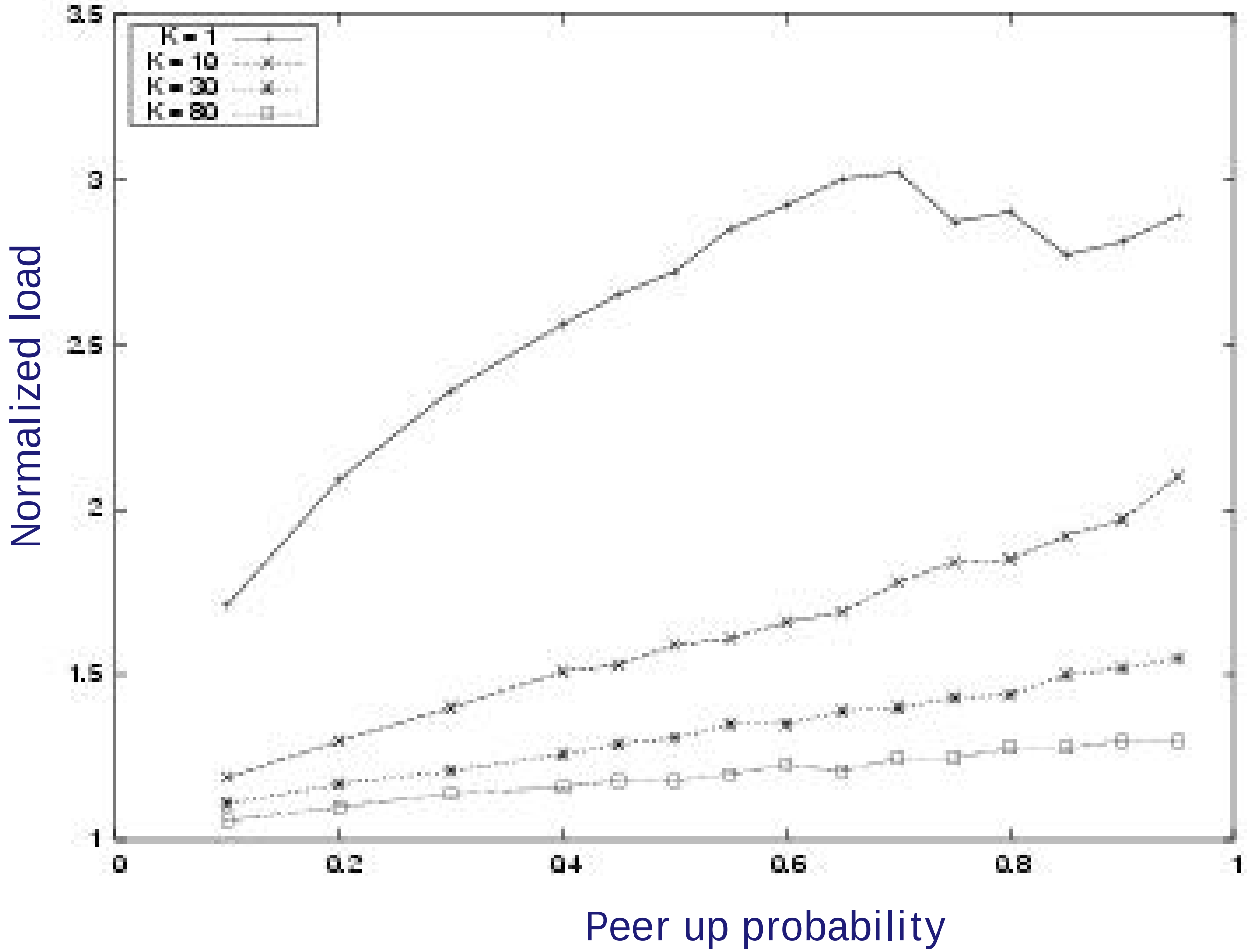
1. Make everyone do the same amount of work?
   - But: Peers might be heterogeneous
2. Allow individual peers to determine their own load?
   - Problem: Too much refused traffic hurts performance

- Two approaches:
  - Fragmentation
  - Overflow

Load Balancing: Solutions

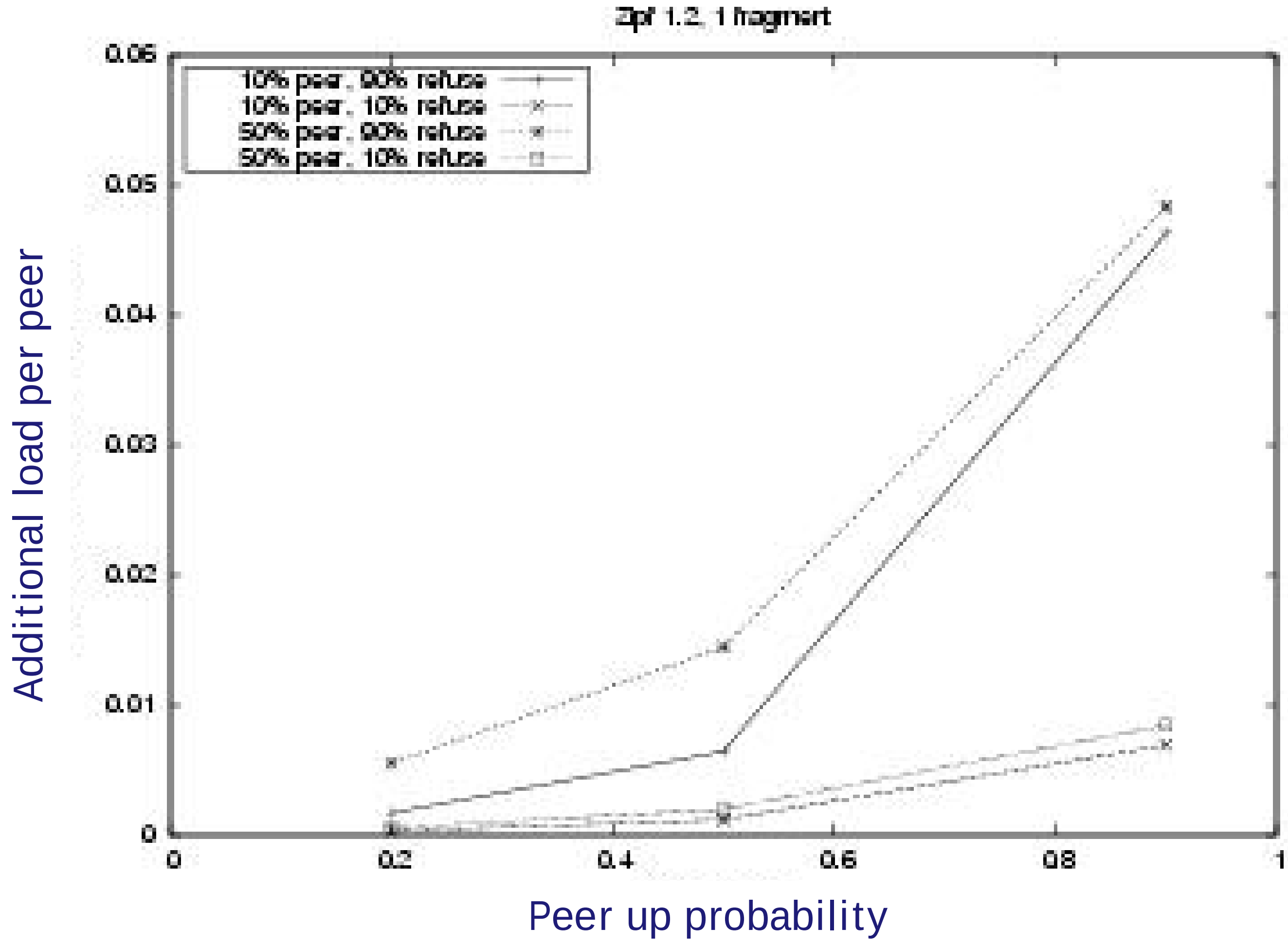
- Fragmentation
  - Idea: Divide each object into chunks, store chunks individually
  - One chunk is much smaller than a file, hence load is balanced better, since chunks are stored on different peers
  - Achieves overall load balancing (goal 1 from above)
- Overflow
  - Idea: Allow peers to refuse requests
  - Request passed on to the next winner (eventually to outside)
  - Allows a peer to decide how much traffic to handle
  - Achieves goal number 2 from above
- Fragmentation + Overflow
  - Use both approaches

Load Balancing: Fragmentation



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

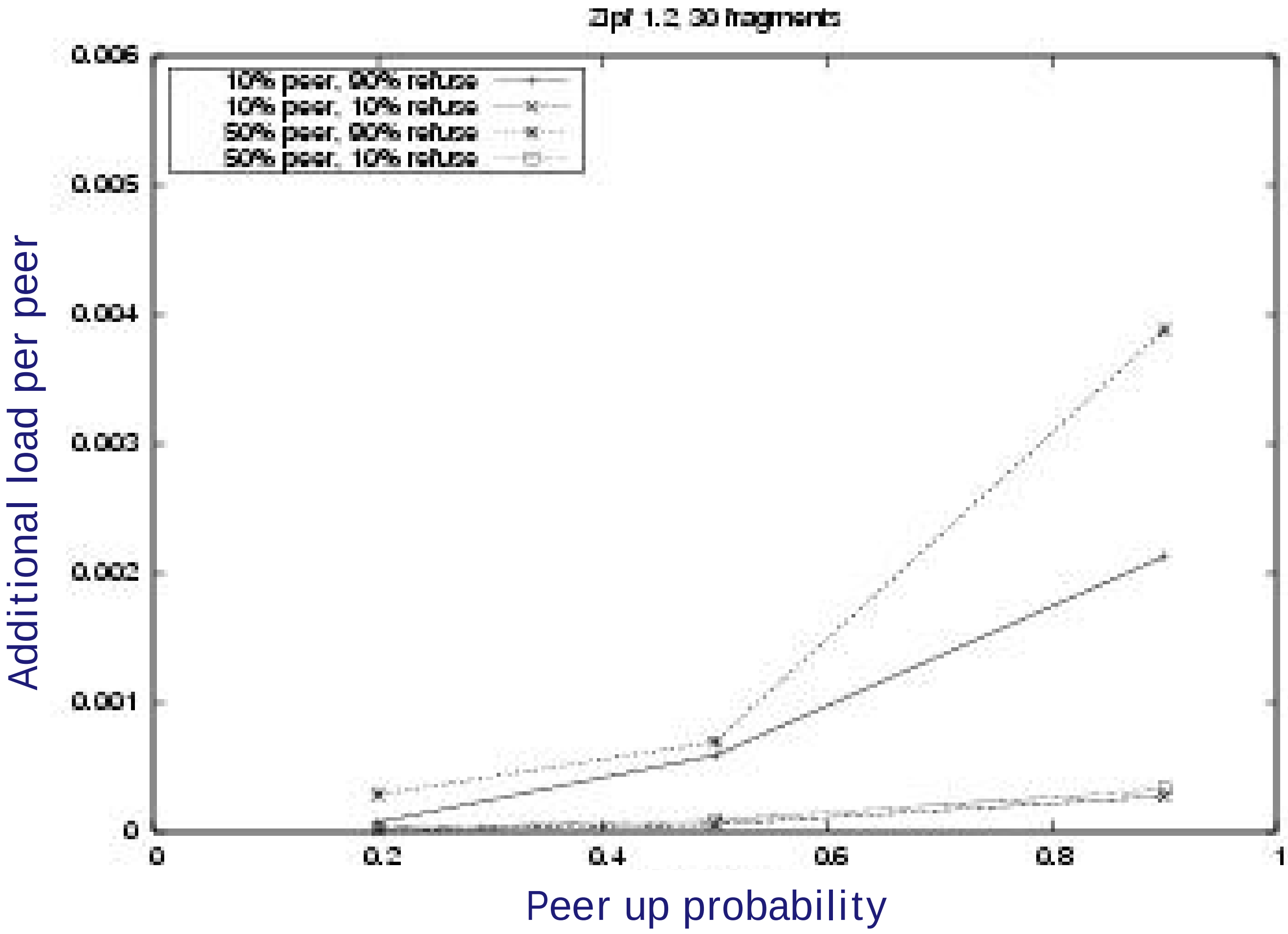
Load Balancing: Overflow



- Overflow with 1 chunk
- Different amounts of refused traffic

- Worst case: 5% additional load for each peer

Fragmentation + Overflow



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

Overflow: Refused Traffic

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

Load Balancing: Summary

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

Chapter Summary

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