Peer-to-Peer and Cloud Computing

Jörg Hähner

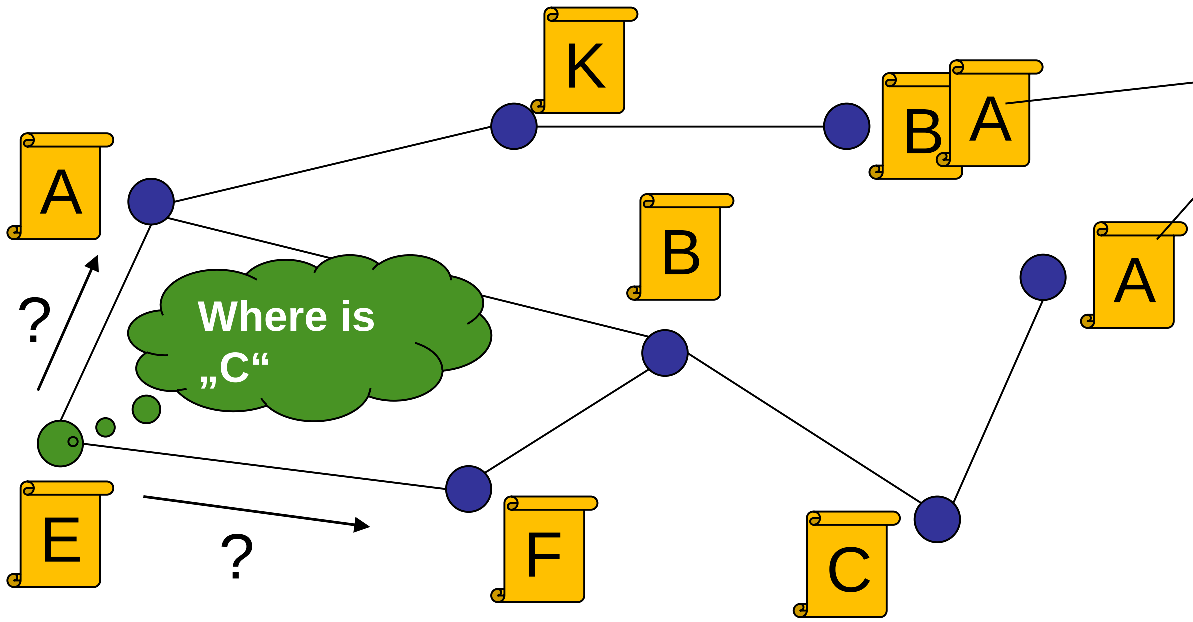
Part 5:

Unstructured P2P systems

- Introduction
- Blind Search
- Informed Search
- Caching and Replication
- Example: Gnutella
- Conclusion

Unstructured P2P systems

- Topology of overlay network is random and independent of content (random graph, small-world network, scale-free network)
  - Neighbours of a peer may store any content
  - Network does not control content placement



Content can be replicated (users may search for a certain number of object copies)

Search in unstructured P2P Systems (2)

Addressing of content

- Anything that can be mapped *locally* to content stored by peer.
  → Typically: set of keywords

Small-world network

- We already know:
  Short path to any peer exists with high probability!
- How to find these paths efficiently in an unstructured P2P network?

What are suitable search algorithms for unstructured systems?

Performance Aspects of Search Algorithms

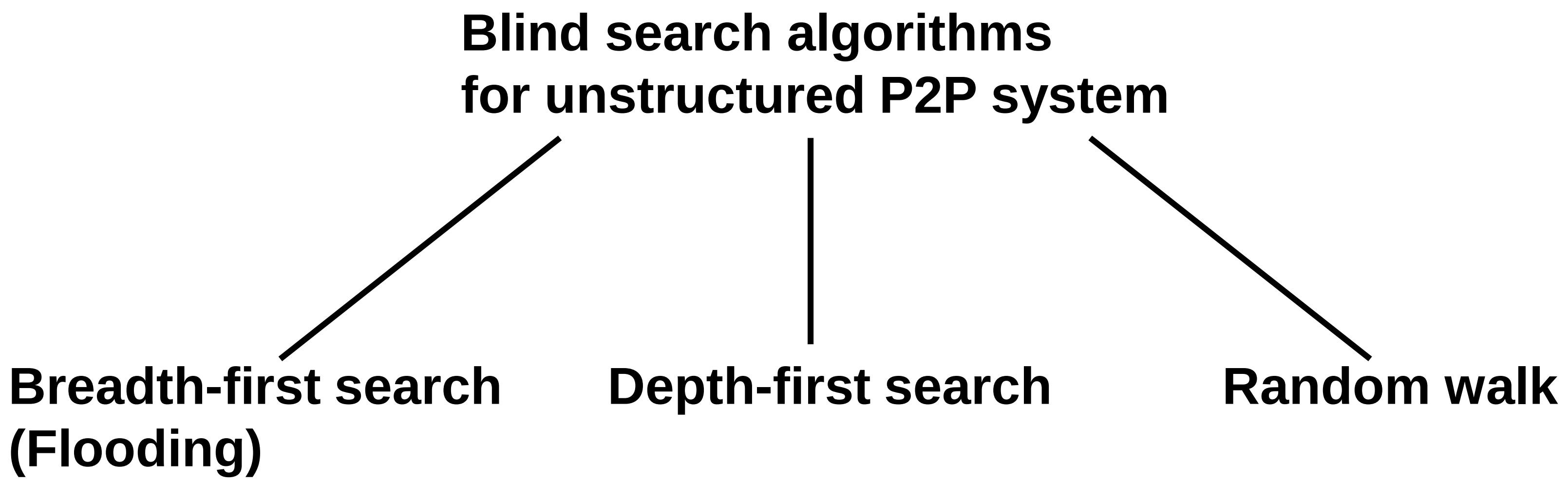
- User aspects:
  - Success rate:
    Does the search algorithm find a requested object copy?
  - Search latency:
    How many steps / hops are required until the search terminates?

- Network aspects:
  - Overhead:
    How many redundant query message copies are generated?
  - Load:
    How many peers are visited during the search?

Search Algorithms for Unstructured Networks

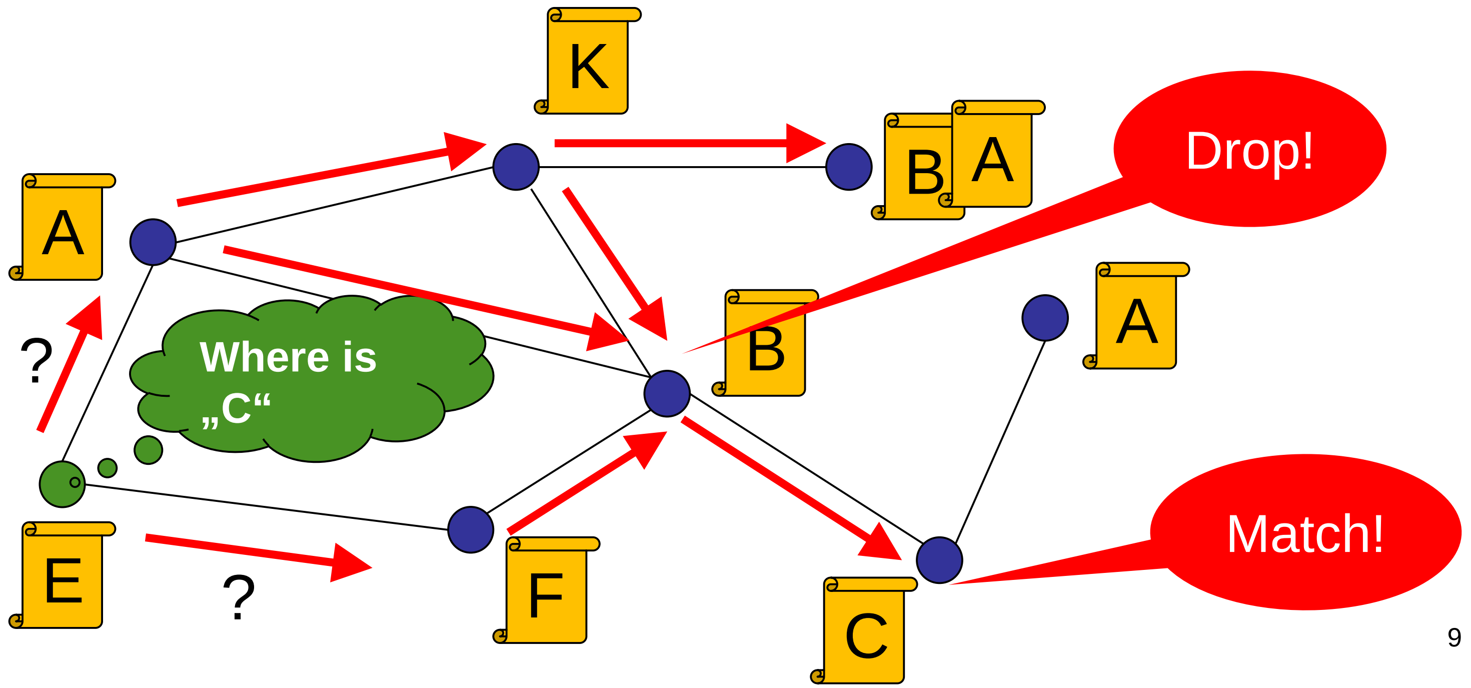
- Blind search
  - Peers have no information about object location.
  - Search solely based on local connectivity information (neighbours).
- Informed search
  - Peers have local information (metadata) related to object location.
  - Metadata assists in search for objects.
    → Peer uses heuristic to select “good” next hop among his neighbours.

Taxonomy of Blind Search Algorithms


Blind search algorithms for unstructured P2P system
Breadth-first search (Flooding)
Depth-first search
Random walk

Plain Breadth-first Search

- Each node n forwards incoming message to all neighbours:
  - Except for the node, from which message has been received.
  - Except for messages that n has "seen" already.
    → Prevents circles.



Plain Breadth-first Search (2)

Pros of plain BFS:

- High success rate:
  - Finds all available objects (100% recall).
  - Only as long as network is not overloaded and query messages get lost.
- Small delay:
  - Finds shortest path to object(s).
  - Only as long as paths or peers are not overloaded and messages queue up.
- Robust:
  - As long as a path exists, it will be found.
  - Only as long as network is not overloaded and messages have to be dropped.

Plain Breadth-first Search (3)

Cons: High communication overhead and network load

- Search may flood whole network
  - Network may be flooded *even if result is found in few steps*!
  - No mechanism to stop search if result has been found.
- Number of visited nodes grows rapidly.
  - Often even “exponentially” (well, at least very steeply)!
- Unnecessary redundant message copies between peers.
  - Node may receive same message from different neighbours.
  - Especially a problem in clustered networks.

Principles of Scalable Search

Lessons to be learned from plain flooding:

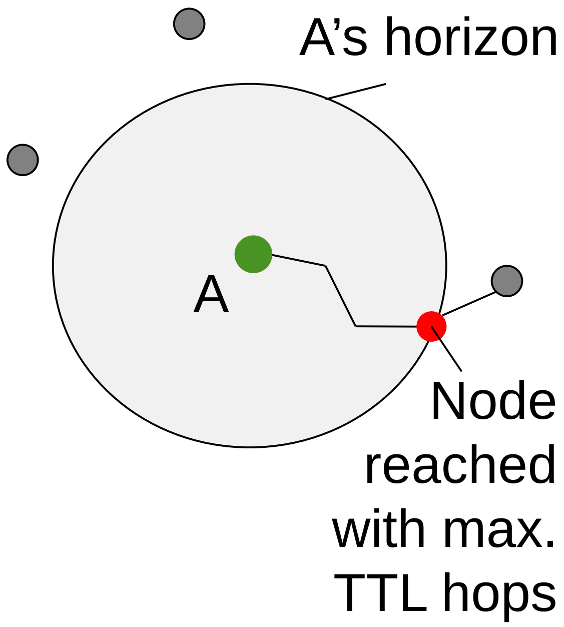
- Adaptive query termination.
  - Query should be stopped when result is found!
- Fine-grained network coverage.
  - Number of nodes visited in each search step should not increase rapidly !
- Minimise message duplicates.
  - Avoid sending the message to peers that have already received it.

These principles should lead to a *scalable search algorithm*:

- Overall query rate can be increased without overloading the network

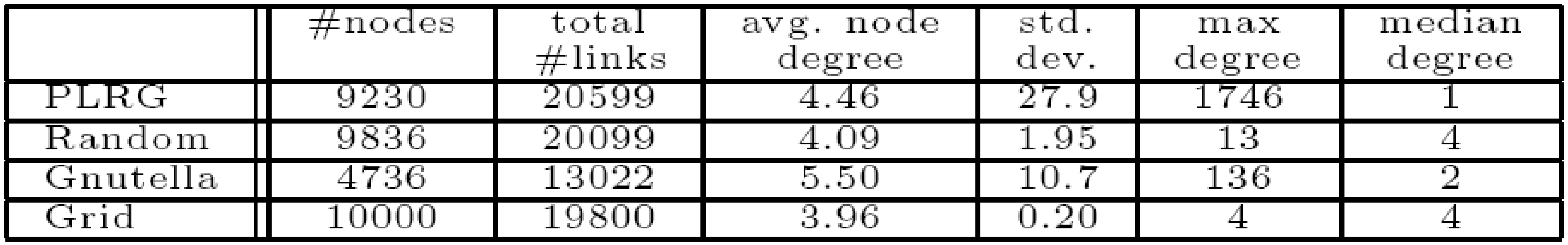
Limited Flooding

- Fixed maximum time to live (TTL) of message.
- Each node decrements TTL on forwarding.
- If TTL = 0, forwarding node drops message.
- TTL leads to a query “horizon”.
  - Query only reaches parts of the network.
- How to chose TTL?
  - TTL small:
    - Search might only reach very limited set of nodes → small recall.
    - Small overhead.
  - TTL great:
    - Search floods large parts of the network → high recall.
    - Great overhead.



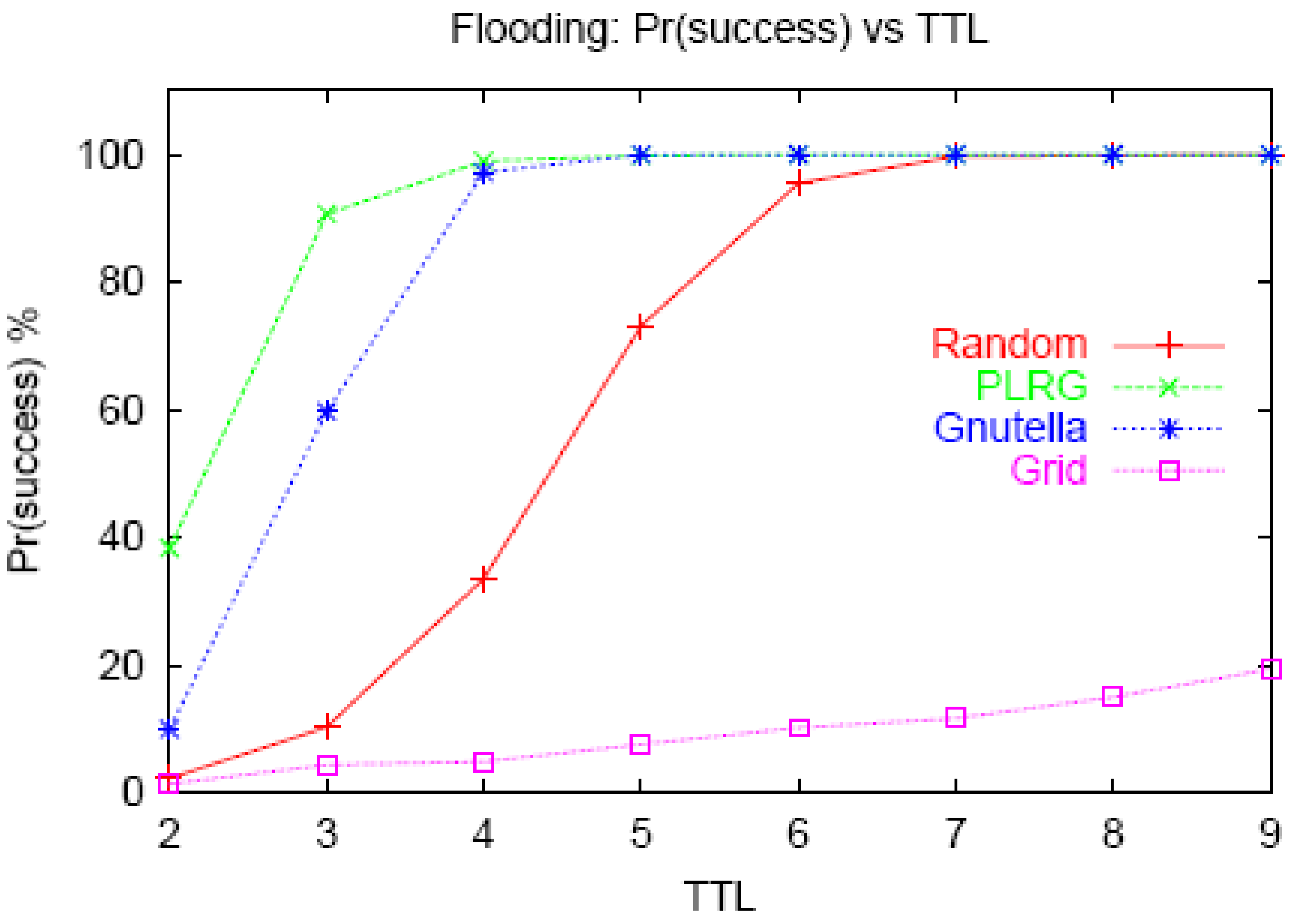
Performance Evaluation of Limited BFS

- Simulations from Lv et al.:
  [Lv, Cao, Cohen, Li, Shenker: *Search and Replication in Unstructured Peer-to-Peer Networks*. Proceedings of the 16th international conference on Supercomputing. 2002]
- Networks
  - Power Law Random Graph (PLRG)
  - Random Graph (Random)
  - Gnutella Graph from 2000 (Gnutella): shows power-law property
  - Regular graph (Grid)



	#nodes	total #links	avg. node degree	std. dev.	max degree	median degree
PLRG	9230	20599	4.46	27.9	1746	1
Random	9836	20099	4.09	1.95	13	4
Gnutella	4736	13022	5.50	10.7	136	2
Grid	10000	19800	3.96	0.20	4	4

Evaluation: Success Rate



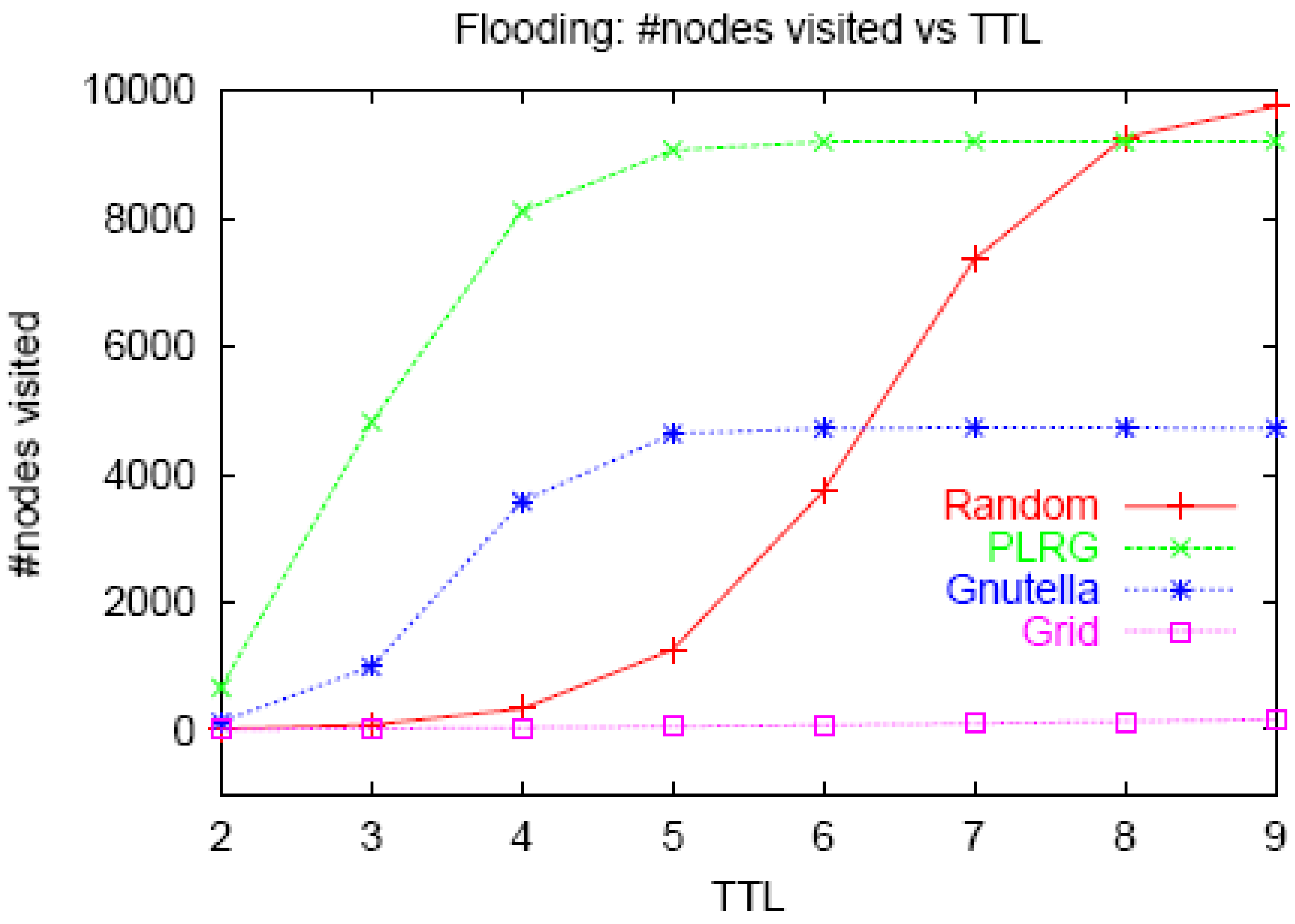
Simulation setup:

- Object replicated at 0.125% of nodes.
- Performance Metric: Success rate Pr(success)
  - Probability of finding object before search terminates.

Result:

- Power-law graphs find objects with small TTL
- Random graph needs slightly more hops
- Regular graph needs much more hops

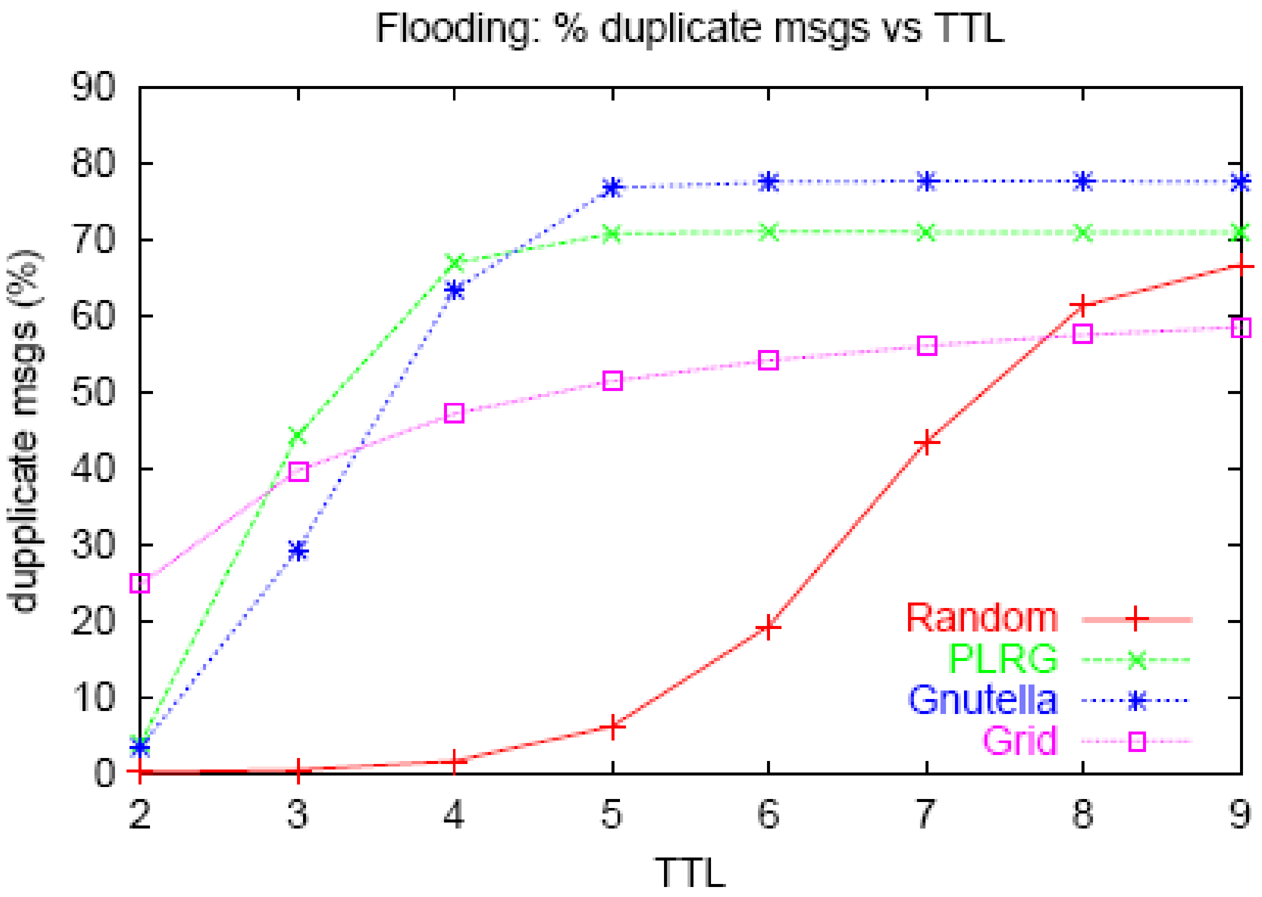
Evaluation: Coverage



- Performance metric:
  - number of unique nodes visited.
- Results:
  - Power-law graphs: message spreads rapidly.
  - Random graph: message spreads slower.
  - Regular graph: slow coverage.

Evaluation: Overhead

- Metric: $r_{duplicate} = \frac{\#messages\ sent\ - \#nodes\ visited}{\#messages\ sent}$



- Result:
  - Power-law graphs: Overhead increases fast.
  - Random graph: Overhead increases much slower.

Properties of Limited Flooding

- Problem of choosing suitable TTL:
  - Small for popular (highly replicated) objects, high for rare objects.
  - Success rate depends on suitable TTL.

  → No adaptive query termination!
- Fast network coverage even for small TTL:
  - Tradeoff between network load and success rate.

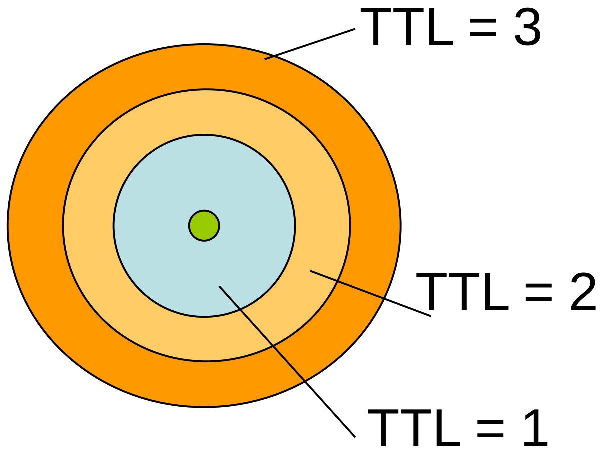
  → No fine-grained network coverage!
- Leads to significant overhead:
  - Very high for power-law graphs, high for random graphs.

  → Does not avoid redundant message copies!

Expanding Ring Search

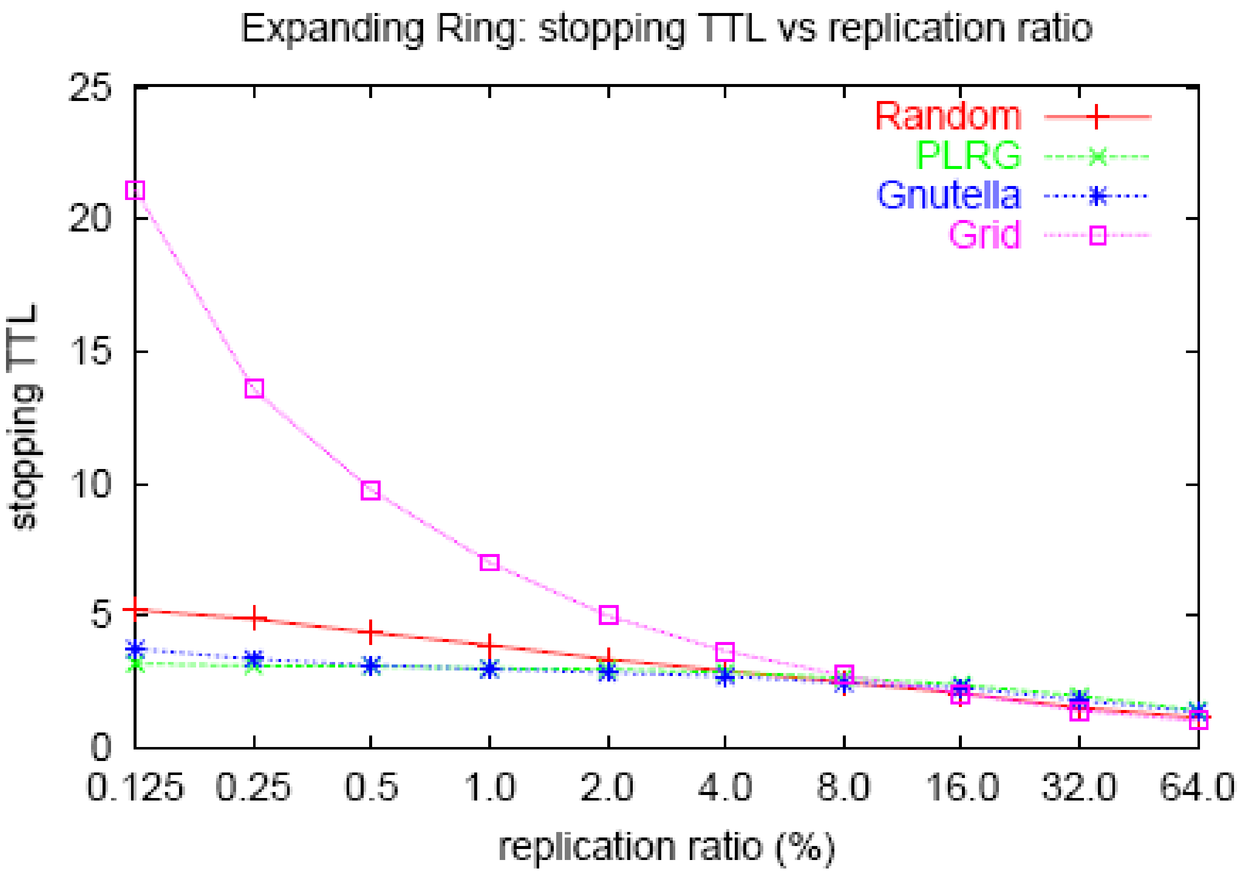
- Query horizon is increased gradually:

```
TTL = 1
do
        send query
        collect results
        increase TTL
until ((number of results >= required number of results) or (TTL > MaxTTL))
```



- Variant: Iterative deepening
  - TTL is increased in steps $\geq$ 1, e.g., {1,5,10}
  - Only nodes beyond old horizon have to process query
    - Nodes remember query for some time
    - Node that has already processed query just forwards it

Performance of Expanding Ring Search



- Stopping TTL:
  - radius of ring required to find an object.
  - Stopping TTL decreases as replication ratio increases.
- Note:
  - even a small decrease might reduce the overhead significantly!
  - Number of visited hops grows rapidly!

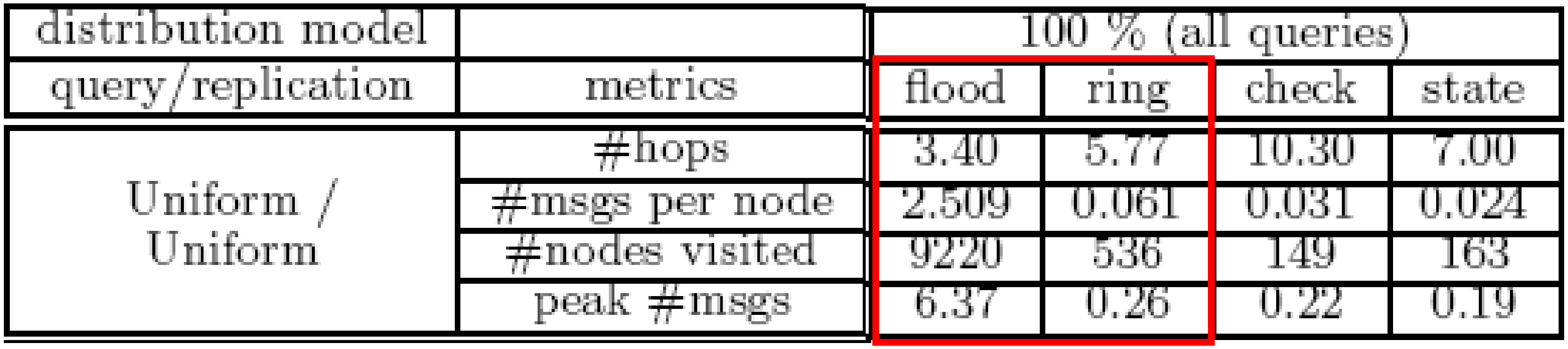
Performance Comparison of Limited BFS & ERS

Simulation parameters:

- Replication ratio: 1.0%
- TTL limit for Limited BFS: 8 (pessimistic)
- Random Graph.

Results:

- Overhead of ERS significantly smaller than Limited BFS
- Latency of ERS slightly greater than latency of Limited BFS



distribution model		100 % (all queries)			
query/replication	metrics	flood	ring	check	state
Uniform / Uniform	#hops	3.40	5.77	10.30	7.00
	#msgs per node	2.509	0.061	0.031	0.024
	#nodes visited	9220	536	149	163
	peak #msgs	6.37	0.26	0.22	0.19

Properties of Expanding Ring Search

Pros:

- Expanding ring search solves problem of choosing TTL.
- Popular items are found rapidly.
- Unpopular items are found if MaxTTL is sufficiently large.
- Overhead & load are reduced significantly *compared to Limited BFS.*

Cons:

- Delay of finding objects increases.
  → n searches instead of one.
- Network coverage and load still high.
  → Number of nodes visited in each step stills grows rapidly.
- Does not avoid redundant message copies.

Randomised Breadth-first Search

- Message is forwarded to neighbour with a given probability $p_{threshold}$:

```
on receiving message m from neighbour n do
    for each neighbour $n_i$ do
        p := random number $\in [0,1]$
        if $n_i \neq n$ and $p < p_{threshold}$ then
            forward m to $n_i$
        fi
    done
done
```

- $p_{threshold}$ can be chosen such that statistically …
  - … duplicate messages are eliminated.
  - … connectivity is preserved.

Properties of Randomised BFS

Pros

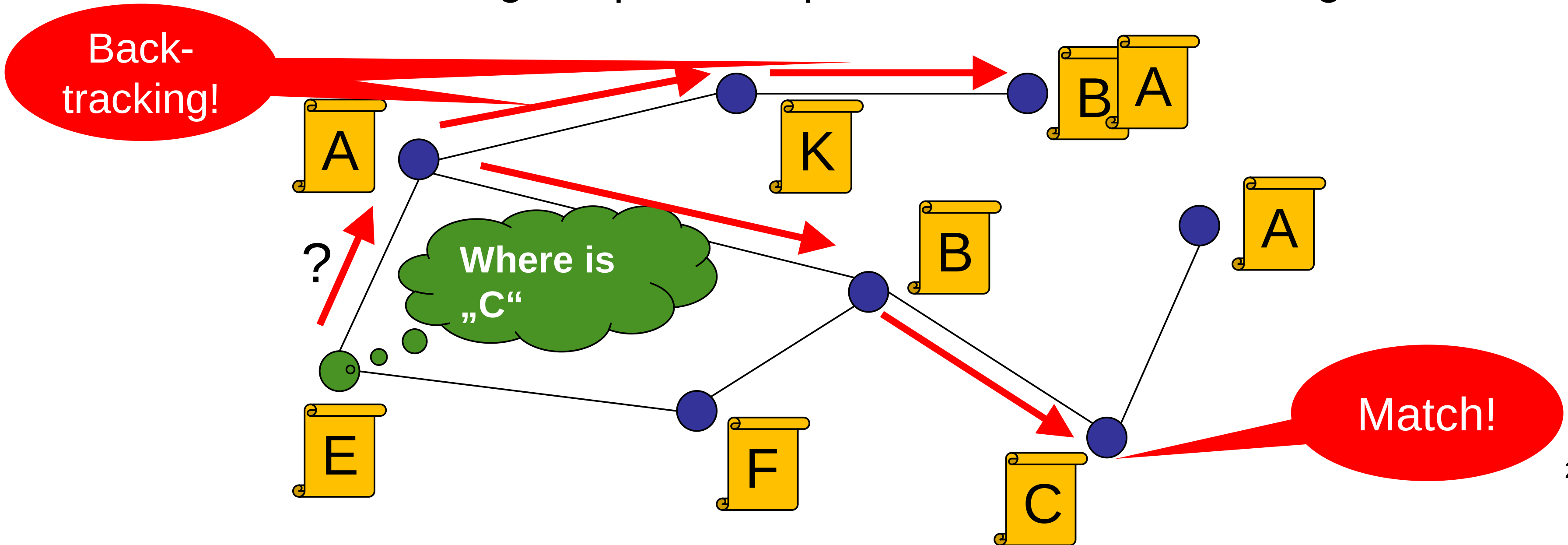
- Reduces overhead significantly.
  → Avoids redundant message copies!

Cons:

- Network coverage still high:
  - Number of nodes visited in each step still grows fast.
- How to chose tuning parameter $p_{threshold}$?
  - Too small: query does not reach all peers → low success rate.
  - Too great: too many message copies → high overhead.
  - Optimal $p_{threshold}$ can be calculated for model graphs.
  - Can $p_{threshold}$ be calculated efficiently for real dynamic graphs?

Depth-first Search

- Each node forwards message to all neighbours sequentially:
  - Only one query message per query in the network.
- When message reaches dead-end: backtracking!
  - Dead-end: No more unvisited neighbours.
  - Return message to previous peer that sent the message.



Properties of Depth First Search

- High success rate.
  - Query reaches all nodes.
- High latency.
  - No parallel search like for breadth-first strategies!
- Small network coverage.

Random Walk

- Each peer forwards the message to *one* randomly chosen neighbour:

```
On receiving message m from neighbour n do
    chose neighbour $n_{next}$ randomly
    forward m to $n_{next}$
done
```

- One query message is one random walker.
- Sender can start $k \geq 1$ random walkers simultaneously.
- k random walkers reach approx. the same number of nodes in i steps as one random walker in k x i steps.
  - Reduces query latency by factor 1/k, increases load by factor k.

Random Walk (2)

- Search termination:
  - Limited TTL → Limit the path length of a random walker!
  - Checking → Walker periodically checks with requester before going to next hop.
- Optimisation:
  - Peers keep state of walks:
    - ID of query.
    - Neighbours to which query has been forwarded already.
    - Query is never forwarded to same neighbour twice by same peer.

Performance Comparison: Random Walk & BFS

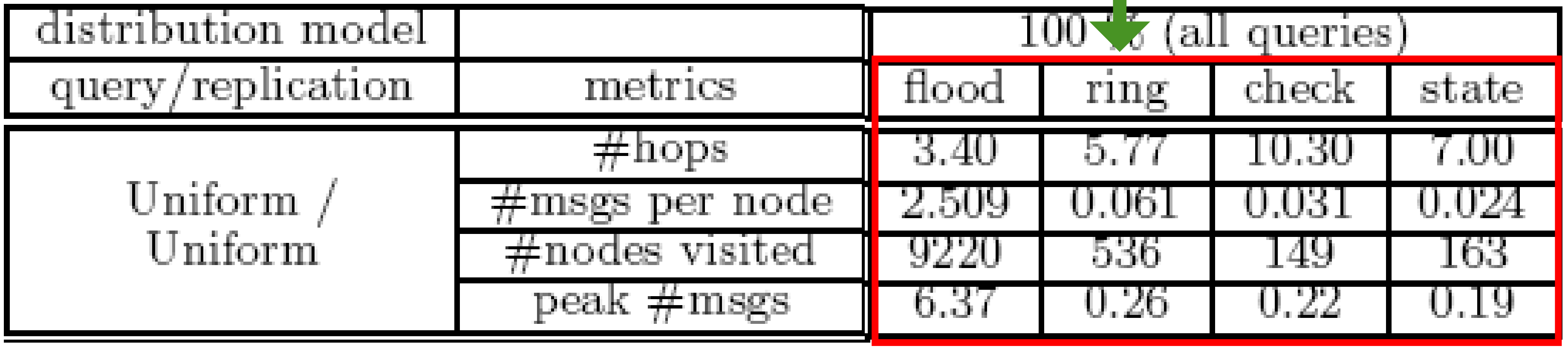
Simulation parameters:

- Replication ratio: 1.0%
- 32 Random walkers
- Random Graph.

Results:

- Search latency of random walk greater than BFS.
- Network load of random walk significantly smaller than BFS.

ERS

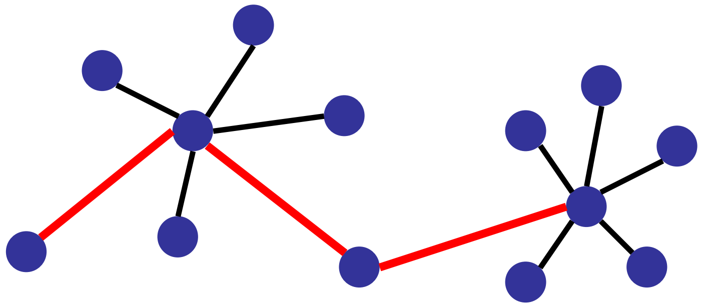


distribution model		100 % (all queries)			
query/replication	metrics	flood	ring	check	state
Uniform / Uniform	#hops	3.40	5.77	10.30	7.00
	#msgs per node	2.509	0.061	0.031	0.024
	#nodes visited	9220	536	149	163
	peak #msgs	6.37	0.26	0.22	0.19

Note: the number of visited nodes increases with the state, since no duplicate nodes are visited anymore.

Hybrid Blind Search

- Hybrid Blind Search:
  - Random Walk + Flooding



- Process:
  - Do a random walk.
  - Each node visited by walker does limited flooding.
    → Shallow flooding: small TTL (e.g. TTL=1).
  - Reduces latency at the expense of overhead:
    - Delay savings especially in power-law graphs.
    - High-degree nodes can search many neighbours in one step.

- Introduction
- Blind Search
- Informed Search
- Caching and Replication
- Example: Gnutella
- Conclusion

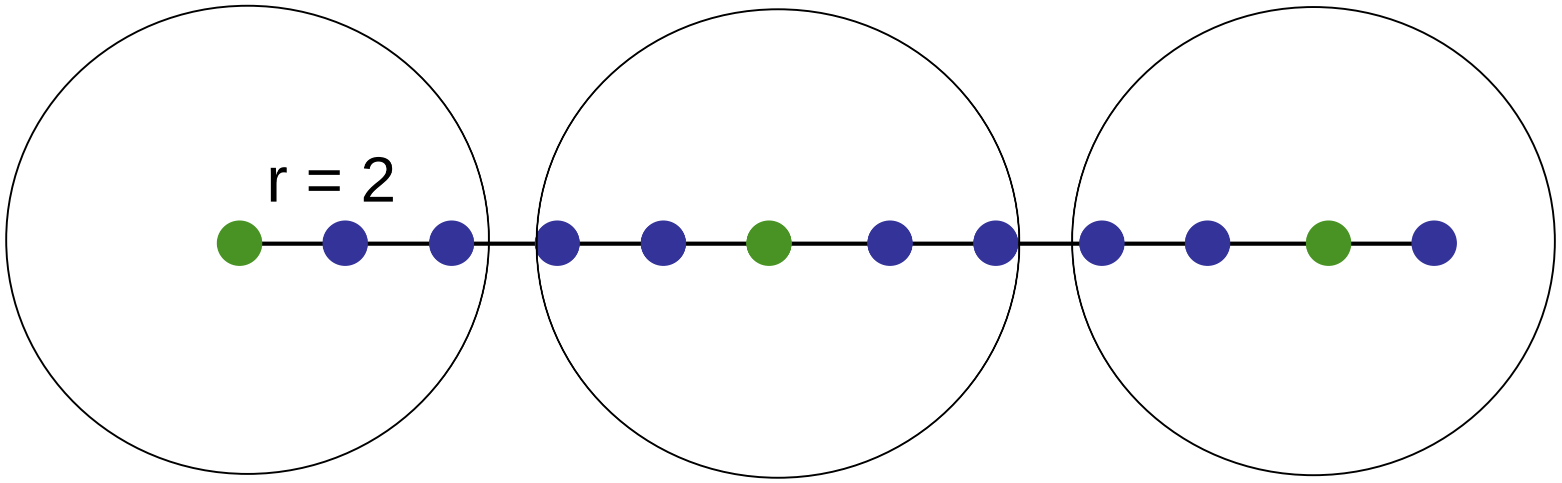
Directed Breadth-first Search

- Variant of limited breadth-first search.
- Each node forwards queries to a subset of its neighbours.
- Next peers are selected based on some heuristic:
  - Peer returned most results for previous queries.
  - Peer returned results with smallest hop count.
  - Peer with shortest message queue.
  - Peer that forwarded the largest number of messages.
  - Peer with highest degree.
  - Peer that answered similar queries successfully.

Local Index

Local Index:

- Each peer keeps a local index of the objects provided by peers within network radius r:
  - Peer can answer queries on behalf of the indexed peers.
  - Peer announces its content by flooding with TTL = r.
- Search: Breadth-first
  - Radius of search is increased in steps of 2r+1 hops.
  - Only peers at distance (2r+1)i to requestor process query.



Properties of Local Index

- Same query result as for ordinary breadth-first search.
- Processing load of peers decreases.
  - Not every peer has to process the query.
- Higher communication cost for join operations:
  - Peer floods summary of its content within radius r.
  - Problem in highly dynamic systems!

Adaptive Probabilistic Search

- k independent random walkers.
- Each peer has local index: $(o, n_i) \rightarrow v$
  - o: object that peer requested or forwarded query for.
  - n_i: i-th neighbour.
  - v: value defining relative probability that query for o should be forwarded to n_i.
- Probabilistic forwarding of query for object o:
  - If peer cannot answer query, it forwards it to the neighbour with the probability defined by (o, n_i).
  - Example: $(o, n_1) = 20$, $(o, n_2) = 30$, $(o, n_3) = 15$
    $\rightarrow$ Message forwarded to n_2 with probability $30/(20+30+15)$.

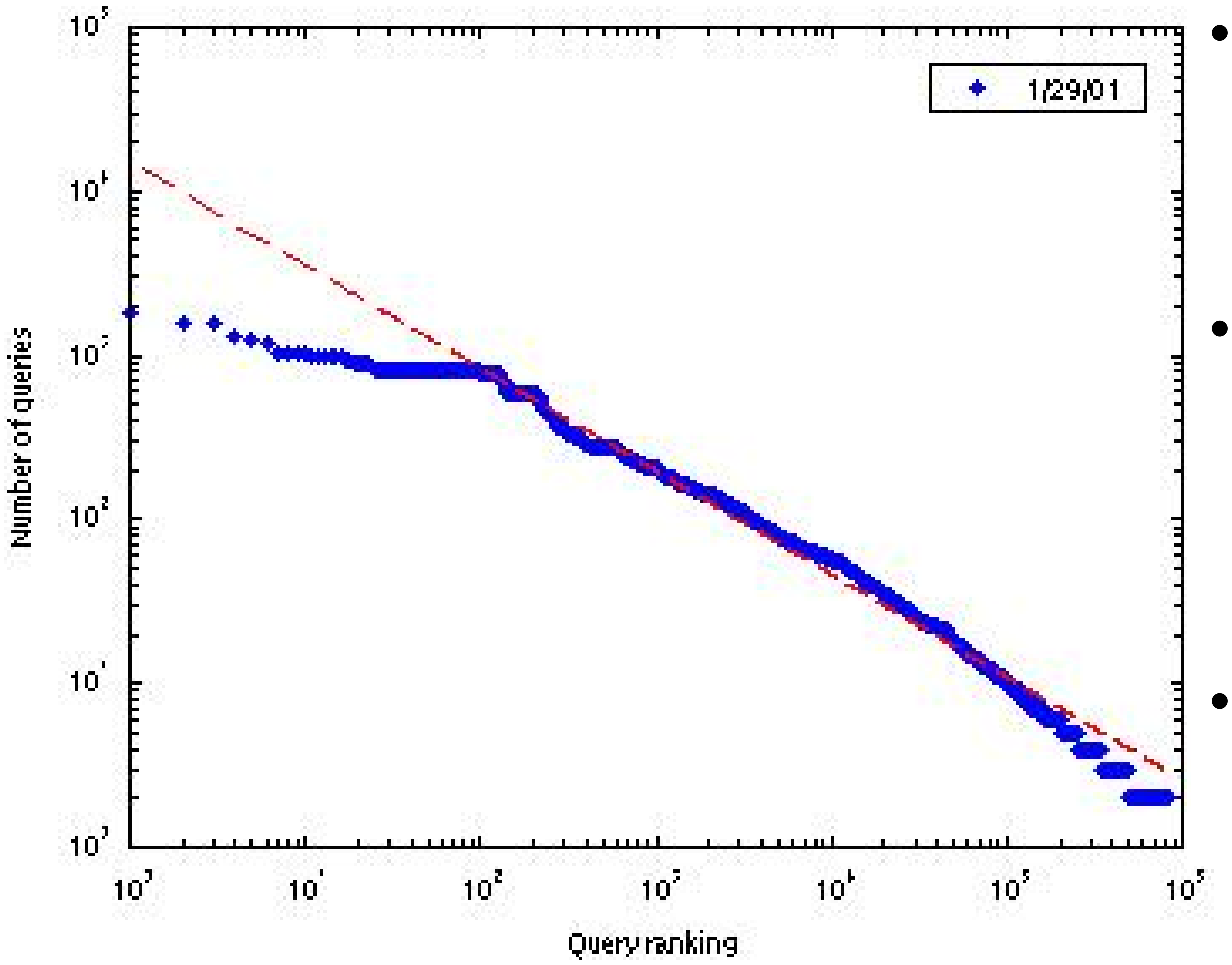
Adaptive Probabilistic Search (2)

- Peer “learns“ probabilities from successful and unsuccessful queries.
- Pessimistic approach:
  - Peer assumes that query will fail.
  - Peer decreases (o, n_i) on forwarding query to $n_{i.}$
  - When query terminates successfully:
    - Terminating peer sends an update message back via reverse path.
    - Peers on reverse path increase the value of (o, n_i) again.
- Optimistic approach:
  - Peer increases value on forwarding.
  - Update sent via reverse path when query fails.
  - Peer decreases value again.

Properties of Adaptive Probabilistic Search

- Smaller query latency:
  - Objects are found faster than using ordinary random walk.
- Increased success rate:
  - Success rate of finding objects increases compared to ordinary random walk with same TTL.
- Only small overhead for update messages:
  - Result must be sent via reverse path.
- No overhead when peers join and leave network:
  - Highly dynamic network is not a problem.

Query Distribution in Gnutella

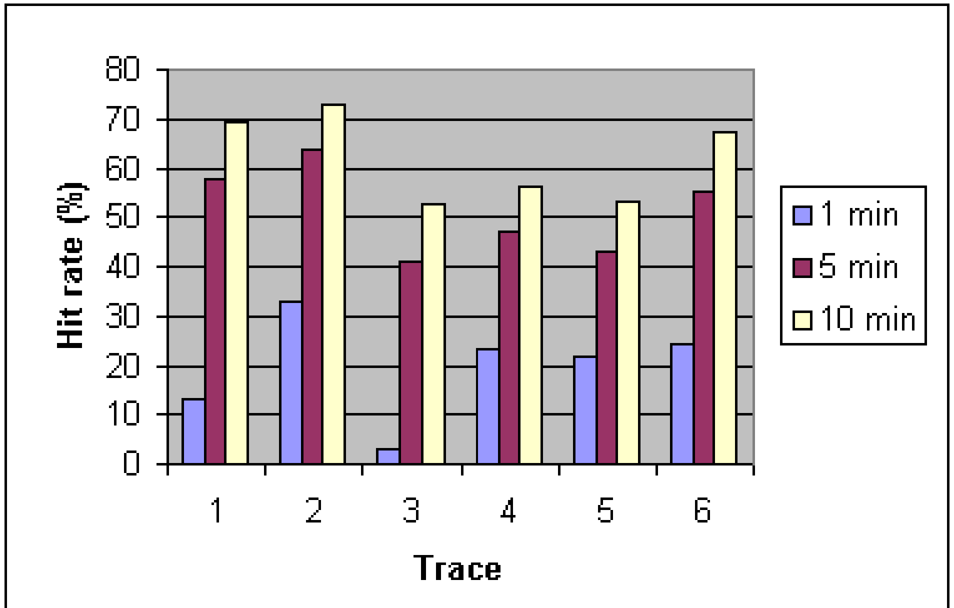


[source: Sripanidkulchai: *The popularity of Gnutella queries and its implications on scalability.*]

- Few very popular documents
  - Almost equally popular
- Power-law distribution of less popular queries
  - Probability of i-th most popular query
  - Similar to web pages
- Can caching be beneficial for Gnutella?

Caching of Query Results

- Peers cache query results:
  - Assumption: result sent via reverse path of query.
- Cache entries are associated with time-outs:
  - Entry deleted when time-out expires.
  - Reduces stale entries in cache due to dynamic peers.
- Replacement policy required for limited cache size (e.g. LRU).
- Results:
  - Up to 73% hit rate.
  - Up to 3.7 times traffic reduction.



Replication

- Observation:
  - Highly replicated popular items are found in few steps.
- Idea:
  - Replicate objects at peers to decrease number of peers to be visited in search.
  - Controlled replication: controlled number of replicas.
- Optimum: full replication.
  - Trivial search: every object found with 0 steps.
  - However: peers’ storage (and bandwidth) is limited.
- What are good replication strategies?

Replication (2)

Model:

- Fixed number of peers: n
- Fixed number of objects: m
- Object i replicated at r_i peers, i.e., object i has r_i replicas
- Object i accessed with relative rate q_i $(\sum_i q_i = 1)$
- Fixed total number of replicas: $R = \sum_i r_i$
- Fixed average number of replicas at each peer: $p = R/n$
  - $p < m$, i.e., no full replication possible
- Each search step probes one peer randomly

Replication (3)

Performance metric: average search size

- Average search size of object i: $A_i = n/r_i$
  - Inverse of fraction of peers that have replicas of object i
  - Expected number of search steps to find object i using a random blind search
- Overall average search size: $A = \sum_i q_i A_i = n \sum_i q_i/r_i$
  - Expected number of search steps averaged over all objects
  - Metric for the expected overall network load

Goal: Find a replication strategy that minimises A!

Uniform Replication

- Each object gets the same number of replicas: r_i = R/m
  - Objects are replicated uniformly among peers
- Number of peers replicating object i: r_i = R/m
- Fraction of peers having replicas of object i: r_i/n
- Search size of object i: $A_i = n/r_i = nm/R = m/p$
- Overall average search size:

$$A_{uniform} = \sum_i q_i \frac{m}{p} = \frac{m}{p}$$

Proportional Replication

- Give more replicas to objects that are frequently accessed:
  - Number of replicas of object i proportional to access rate of object i: $r_i = Rq_i$
  - Each object gets its “faire share” according to its importance.
  - Similar to real P2P systems like Gnutella:
    - Popular files are replicated at many peers (but: uncontrolled replication in Gnutella).
- Search size of object i: $A_i = n/r_i = n/(Rq_i)$.
- Overall average search size:

$$A_{proportional} = \sum_i q_i A_i = \frac{n}{R} m = \frac{m}{p} = A_{uniform}$$

Comparison: Uniform / Proportional Replication

- Same overall average search size:
  - Same overall network load.
- Uniform replication:
  - All objects have same average search size.
  - Replica utilisation rate ($U_i = Rq_i/r_i$) proportional to query rate:
    - Rate of requests one replica of object i serves.
    - Metric for load balancing property.
- Proportional replication:
  - Popular items have small average search size.
  - Unpopular items have large average search size:
    - That‘s why Gnutella is good for popular objects but bad for unpopular.
  - All replicas have same utilisation rate (1) $\rightarrow$ perfect load balancing.

Square-root replication:

- Optimal replication strategy with respect to overall network load (i.e. average search size).

$r_i = \lambda\sqrt{q_i}$ with $\lambda = \frac{R}{\sum_i \sqrt{q_i}}$

- Overall average search size:

$$A = \frac{\left(\sum_i \sqrt{q_i}\right)^2}{p}$$

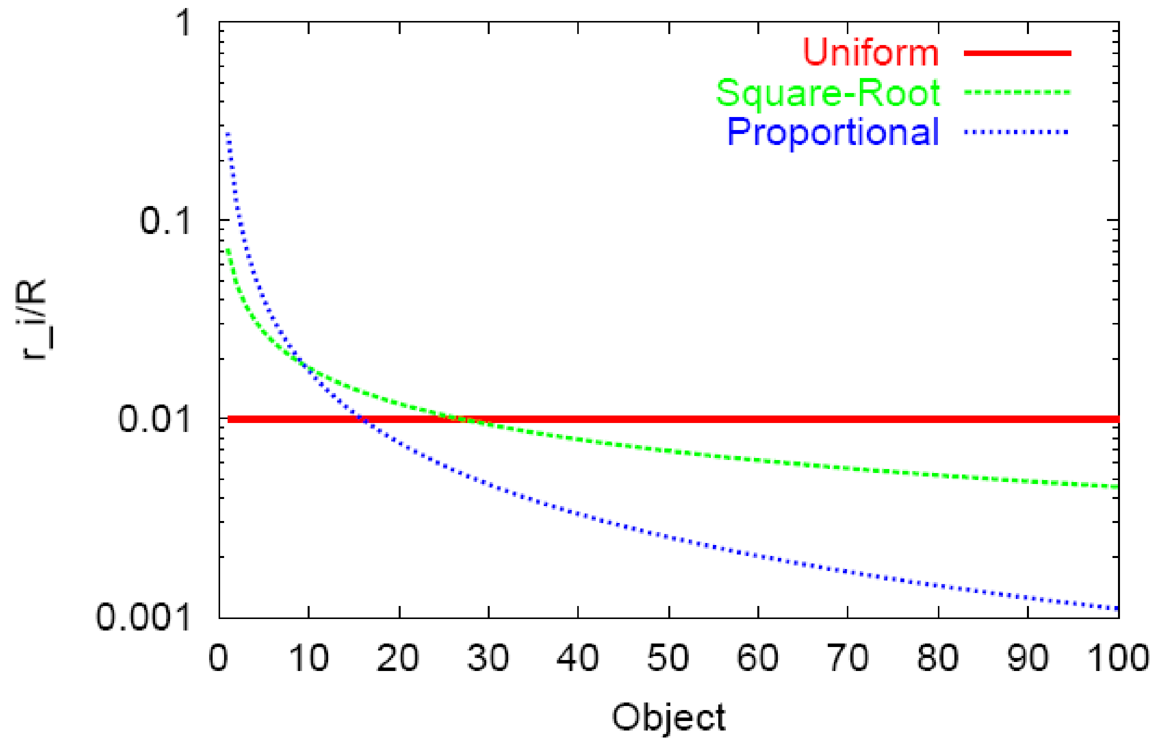
Comparison of Replication Strategies

Assumption: Query popularity distribution follows power-law

- Few very popular queries/objects
- Many almost equally popular objects

Number of replicas

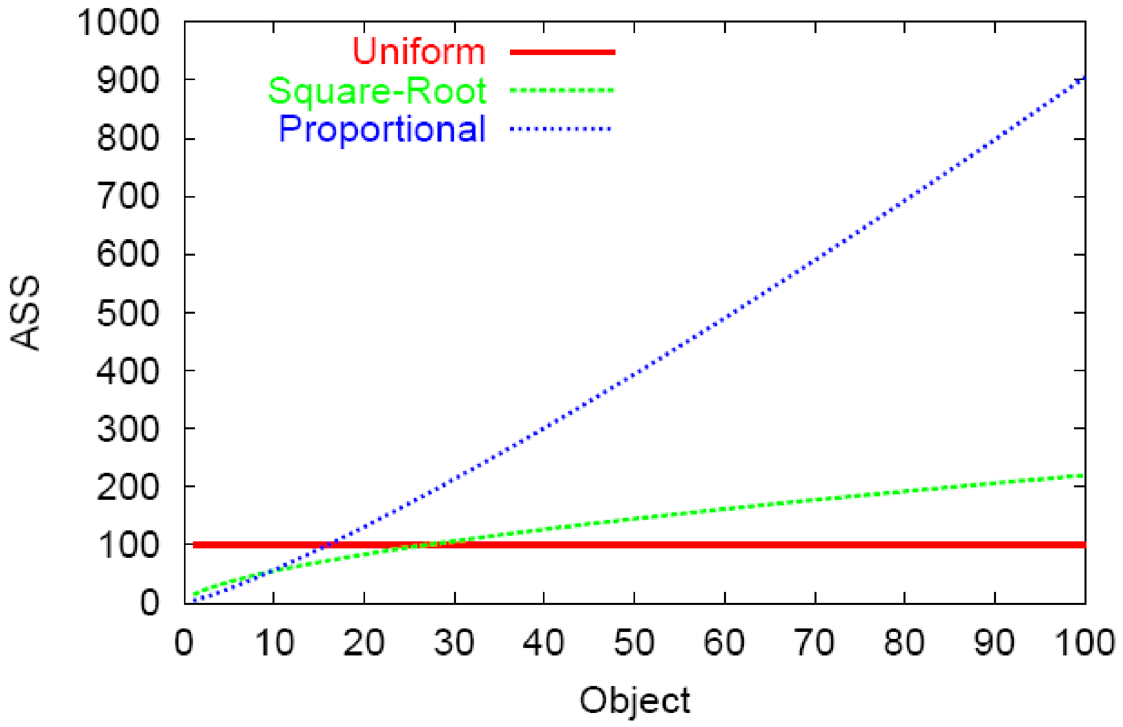
- Uniform and square-root rep.:
  - Majority has more than fair share
- Proportional rep.:
  - Fair-share



Comparison of Replication Strategies (2)

Average search size (ASS = pA_i)

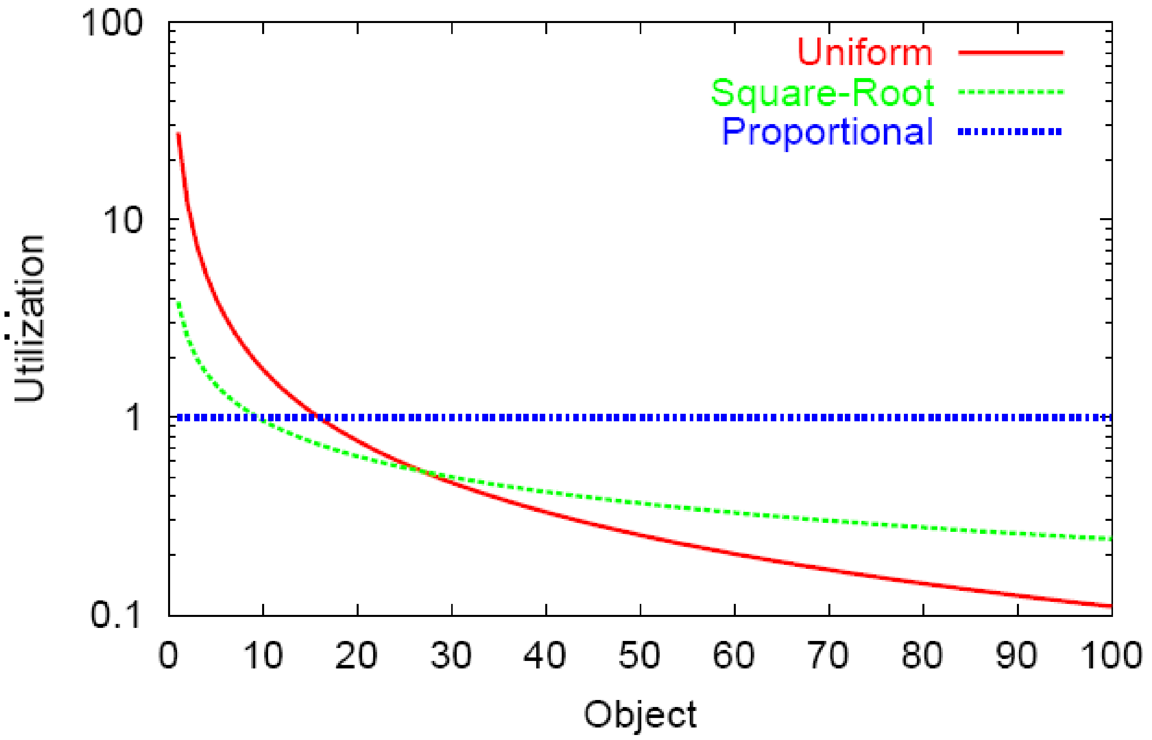
- Uniform replication:
  - All objects have same search size
- Proportional and square-root rep.
  - Popular objects have smaller search size than unpopular
  - Variance smaller for square-root



Comparison of Replication Strategies (3)

Replica utilisation ($U_i = Rq_i/r_i$)

- Proportional replication
  - All object replicas have same utilisation
  - Perfect load balancing
- Uniform and square-root replication:
  - Popular object replicas have higher utilisation than unpopular
  - Variance smaller for square-root



History of Gnutella

Gnutella:

- Decentralised P2P file sharing system.
- May also be used as general “search engine”.

- First version of Gnutella in year 2000:
  - Developed in 14 days by Justin Frankel & Tom Pepper.
    → Winamp authors, Nullsoft company bought by AOL in 1999.
  - Experimental software to share recipes.
  - Download withdrawn by AOL within one day, but software has already been downloaded.
- Gnutella protocol reverse engineered:
  - Lots of open source Gnutella “clients”.

Typical Procedure in Gnutella

- New peer P joins the Gnutella network:
  - Bootstrapping: P finds a node that is already part of the network.
  - Joining the network:
    1. P connects to one other peer through TCP/IP (Handshake).
    2. P discovers further peers of the network.
    3. P establishes further TCP/IP connections to other neighbours.
- Searching:
  1. P sends query for certain object.
  2. Other peers forward message to peers providing object or to peers knowing sources providing this object.
  3. Results (typically > 1) are returned to P via reverse path.
  4. P selects one provider peer and downloads the object.

Boostrapping in Gnutella 0.4

- New peer must know peers that are already part of the network to join it → Bootstrapping mechanisms:

- Word of mouth (used only in early days):
  - Ask some friends via IRC for Gnutella peers.
- Web caches (GWebCache):
  - Web servers providing lists of Gnutella peers via HTTP.
- Local host caches:
  - Peer stores known hosts from last session persistently.
    → Cache populated by replies from caches, pongs, queries.
  - Reduces load on web caches!

Gnutella 0.4 Protocol: Bootstrapping (2)

- UDP Host Caching (UHC):
  - Cache distributed among Gnutella peers.
    → No dedicated web servers required!
  - Peers announce themselves as UHC using Pong messages:
    → Field in Pong message: IP address, optionally DNS name (also works with dynamic IPs).
  - UDP as light-weight connection-less protocol for querying UHC.
    → However, incoming UDP traffic often blocked by firewall.

Gnutella 0.4 Protocol: Message Forwarding

- Message forwarding:
  - Limited flooding.
  - Typical TTL: 7.

Remember the small network number 6?

- Each message has a unique ID:
  - Similar messages with same ID are discarded.
- Replies are returned via reverse path:
  - Peers remember incoming connection message was received from.
  - Replying/forwarding nodes do not know requestor.

MESSAGEHEADER: 23Byte

Msg. ID	**Type**	**TTL**	**Hops**	**Payload Length**
16 Bytes	1 Byte	1 Byte	1 Byte	4 Bytes

Gnutella 0.4 Protocol: Exploring the Network

- Ping/Pong messages for exploring the network:
  - New peer sends Ping message.
    → Ping is forwarded using limited flooding.
  - Peers receiving Ping reply with Pong messages:
    - IP address and port of peer.
    - Number of shared files and kilobytes.
    - Peer can send multiple Pongs for one Ping.
      → Pongs for host cache entries.

PING (Function:0x00) *No Payload*

PONG (Function:0x01)

Port 2 Bytes	**IP Address** 4 Bytes	**Nb. of shared Files** 4 Bytes	**Nb. of Kbytes shared** 4 Bytes

Gnutella 0.4 Protocol: Querying the Network

- Query contains:
  - Minimum download speed required.
  - Search criteria:
    - Text with no exactly specified meaning.
    - String of keywords separated by blanks.
- Query is forwarded using limited flooding.

QUERY (Function:0x80)

Min. Speed / Flags 2 Bytes	**Search Criteria** n Bytes

Gnutella 0.4 Protocol: Querying the Network (2)

Answer: Query Hit message

- Peer with matching content returns Query Hit message
- Number of files matching query
- IP & Port number for download
- Network speed
- Result set:
  - n entries
  - File index, file size, file name

QUERY HIT (Function:0x81)

Nb. of Hits	Port	IP Address	Speed	Result Set	GnodeID
1 Byte	2 Bytes	4 Bytes	1 Byte	n Bytes	16 Bytes

File Download

- Pull:
  - HTTP request to source peer.
    → IP & port number of source included in query hit message.
  - Request contains file index and file name from query hit message.
- Push:
  - Fire-walled peers cannot handle incoming connection request.
    → Pulling files from source via HTTP not possible!
  - Push request sent from requestor to source via reverse path of query hit message.
    → Request routed using GNodeID of query hit message.
  - Source connects to requestor and uploads/pushes file to requestor.

Improvements to Gnutella 0.4

Problems of Gnutella 0.4

- Slow peers:
  - Easily become overloaded → “black holes”.
- High network load caused by limited flooding:
  - Hard to choose good TTL to strike a balance between load and query success rate.
- Further reading: “Why Gnutella Can’t Scale. No, Really.“ by Jordan Ritter (2001)

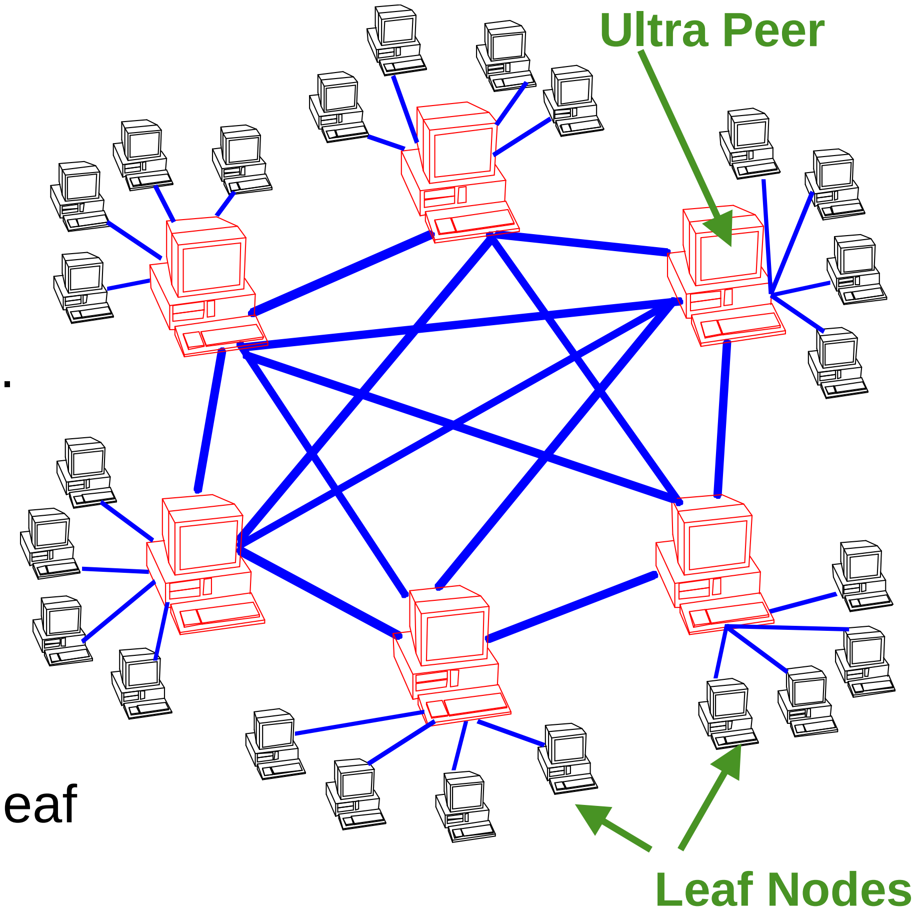
Improvements in Gnutella 0.6

- Hierarchical super P2P network structure:
  - Utilise power of powerful machines and connections.
- Dynamic querying:
  - Dynamically adjust TTL to achieve low network load.

Gnutella 0.6: Hierarchical Super P2P System

Two classes of Gnutella nodes:

- Leaf Nodes:
  - Provide files.
  - Connects to up to 3 ultra peers.
- Ultra Peers:
  - Query routing.
  - Shields leaf nodes from queries that cannot match a leaf's content.
  - Connected to large number of leaf nodes (10 – 100, typically 30).
  - Connected to small number of ultra peers (typically 32).



Gnutella 0.6: Ultra Peer Selection

1. Peer decides to become an ultra peer candidate according to the following criteria:
   - Not fire-walled.
     - Incoming connections possible?
   - Suitable operating system.
     - Large numbers of sockets required.
   - Sufficient bandwidth.
     - > 15KB/s upstream, 10 KB/s downstream.
   - Sufficient RAM.
   - Sufficient CPU speed.
   - Sufficient uptime.
     - At least one hour.

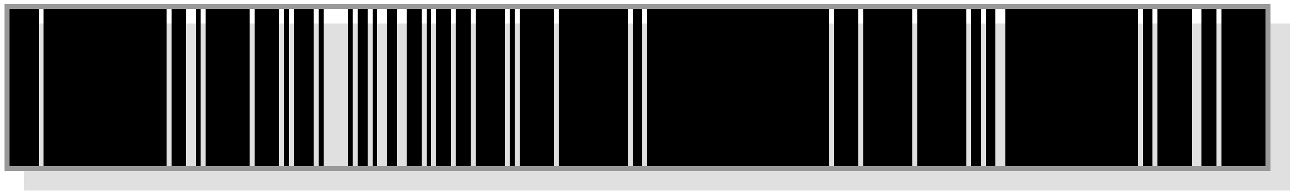
Gnutella 0.6: Ultra Peer Selection (2)

2. Ultra peer candidate *C* tries to connect to another ultra peer *U:*
   - If *U* has too few leaf nodes, it tells *C* to become a leaf of *U*.
     → *C* may try again later to become an ultra peer.
   - If *U* has enough leaf nodes and free ultra peer connections:
     C becomes an ultra peer and connects to *U!*

Gnutella 0.6: Query Routing Protocol

Leaf sends content summary to its ultra peer:

- Break up all resource names into individual words.
- Hash each word to a number in [0,65535].
- Hash each word with trailing 1, 2, 3 characters removed.
  - Simple way to remove plural from words.
- Summary: Bit vector of length 65536.
  - Bits represent the buckets of the hash table.
  - Bit *x* is 1 iff some content of peer is hashed to *x*.



Gnutella 0.6: Query Routing Protocol (2)

Ultra peer to leaf message forwarding:

- Until summary is received from leaf, all incoming queries are forwarded to leaf.
- If Ultra peer receives summary, it becomes the “routing table” for this peer:
  - Ultra peer hashes query keywords to number *x.*
  - If routing table entry *x* of leaf *L* is 1, the query is forwarded to *L.*
  - No guarantee that *L* provides content, but very likely:
    - Guarantee that no peer is excluded that does actually provide the desired content.
  - Guarantee that no peer receives query with no matching content.

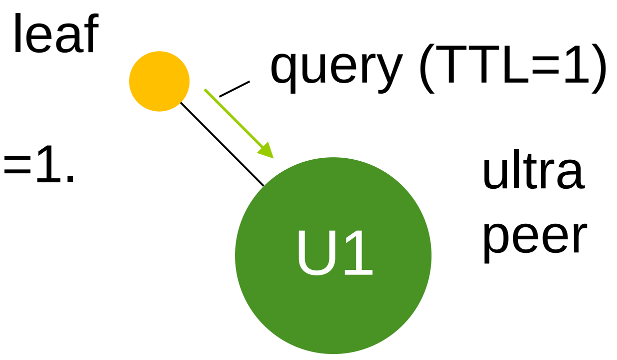
Gnutella 0.6: Query Routing Protocol (3)

Inter Ultra peer message forwarding:

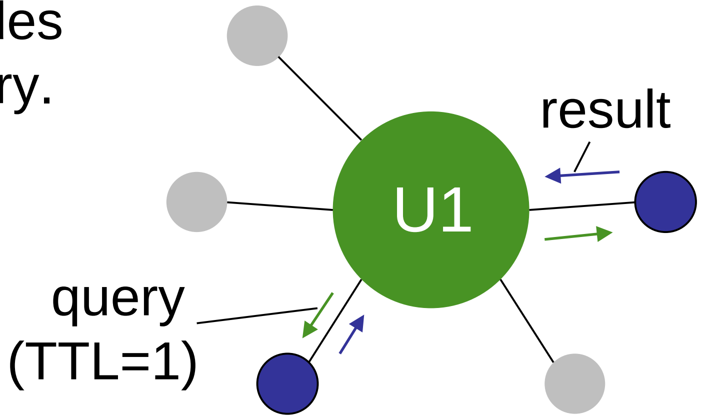
- Blind search: Limited flooding between ultra peers.
- Informed search: Ultra peer calculates supremum of all summaries and sends it to neighbouring ultra peers:
  - Supremum: bit-wise “or” of all routing tables .
  - Example:
    Supremum(00000110,01000000,11000000) = 11000110.
  - Message with TTL <= 2 only has to be forwarded to neighbouring ultra peer *U*, if query hash matches *U*’s query table supremum.

Gnutella 0.6: Dynamic Querying

1. Submitting the query:
   - Leaf sends query to its ultra peer with TTL=1.
   - Query is dead, but ultra peer keeps it and starts dynamic query process.



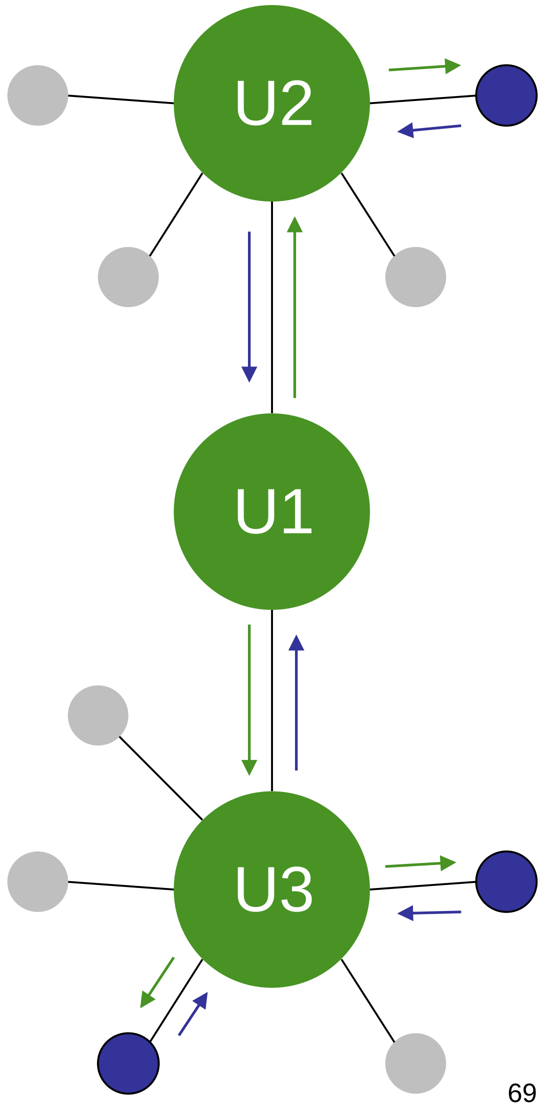
2. Leaf search:
   - Ultra peer sends query to all leaf nodes with matching or no routing table entry.
   - If enough results are returned within timeout interval, search is done.
     → Return the results to the leaf.
   - Otherwise: proceed to step 3.



Gnutella 0.6: Dynamic Querying (2)

3. Probe Query

- Question:
  How popular is the queried object?
  How deep should we query? (TTL?)
- Do a broad shallow probe query:
  - U1 sends query with small TTL to subset of neighbouring ultra peers.
  - Neighbouring ultra peers do a leaf search and return results to U1.
- If enough results are returned within timeout interval, search is done.
  → Return the results to the leaf!
- Otherwise: proceed to step 4.

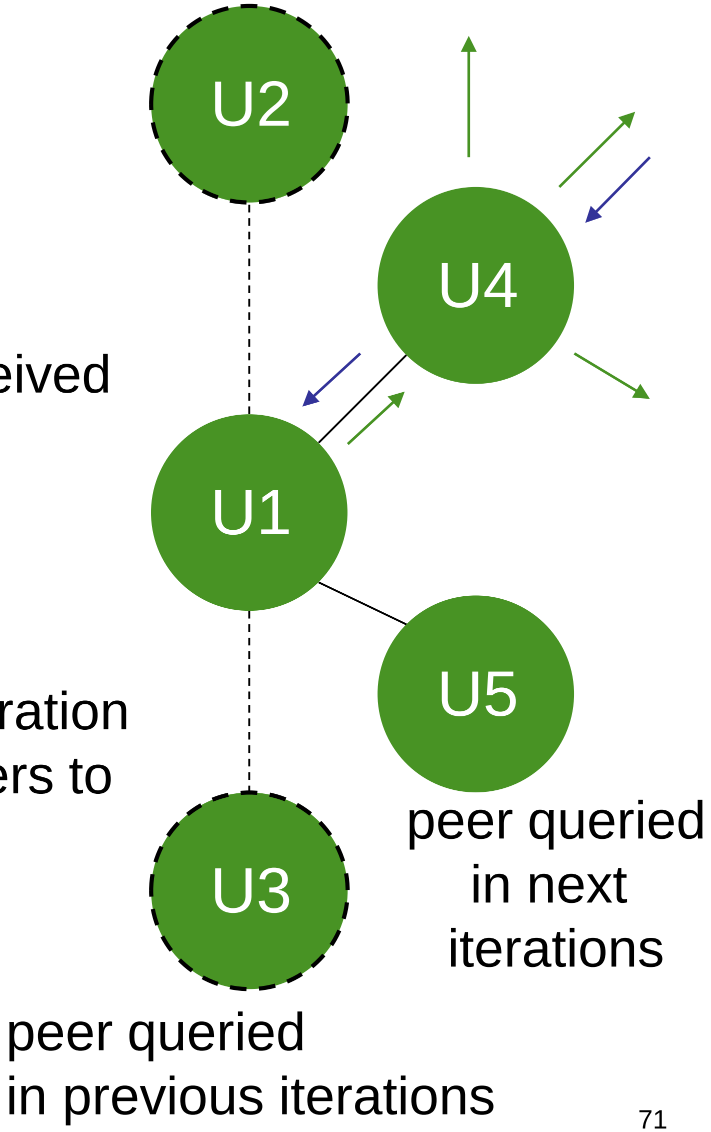


Gnutella 0.6: Dynamic Querying (3)

- Heuristic to determine number of neighbours and TTL of probe query based on neighbour query routing tables:
  - Query blocked by all ultra peer routing tables:
    → Query is unpopular: 3 neighbours, TTL 2.
  - Query matches all ultra peer routing tables:
    → Query very popular: 1 neighbour, TTL 1.

Gnutella 0.6: Dynamic Querying (4)

4. Controlled Flooding:
   - Iteratively, do deep narrow queries.
   - In each iteration, the query is sent to one neighbour of U1 that has not received the query yet:
     - Limited flooding with given TTL.
     - Timeout between successive queries to collect results.
   - The query TTL is adjusted in each iteration to hit the required number of ultra peers to find the required remaining number of objects.


U2
U4
U1
U5
U3
peer queried in next iterations
peer queried in previous iterations

Gnutella 0.6: Dynamic Querying (5)

Calculating the required TTL for each iteration:

- Desired number of results: $r_{desired}$ (typically <= 150)
- Results received so far: $r_{received}$
- Results still required: $r_{required} = r_{desired} - r_{received}$
- Estimated number of ultra peers queried so far: $h_{queried}$
- Expected number of results per ultra peer: $r_h = r_{received} / h_{queried}$
- Required number of ultra peers still to be queried to get the remaining required number of results: $u_{required} = r_{required} / r_h$
- Remaining ultra peer connections of U1 that have not been queried yet: c
- Number of ultra peers to query per connection: $h_{connection} = u_{required} / c$

Gnutella 0.6: Dynamic Querying (6)

Calculation of TTL for next query iteration:

- Degree of ultra peer: k (typically 32)
- TTL for next iteration: minimum t such that message reaches $h_{connection}$ ultra peers
  - Estimated number of ultra peers reached per connection by query message with TTL t:

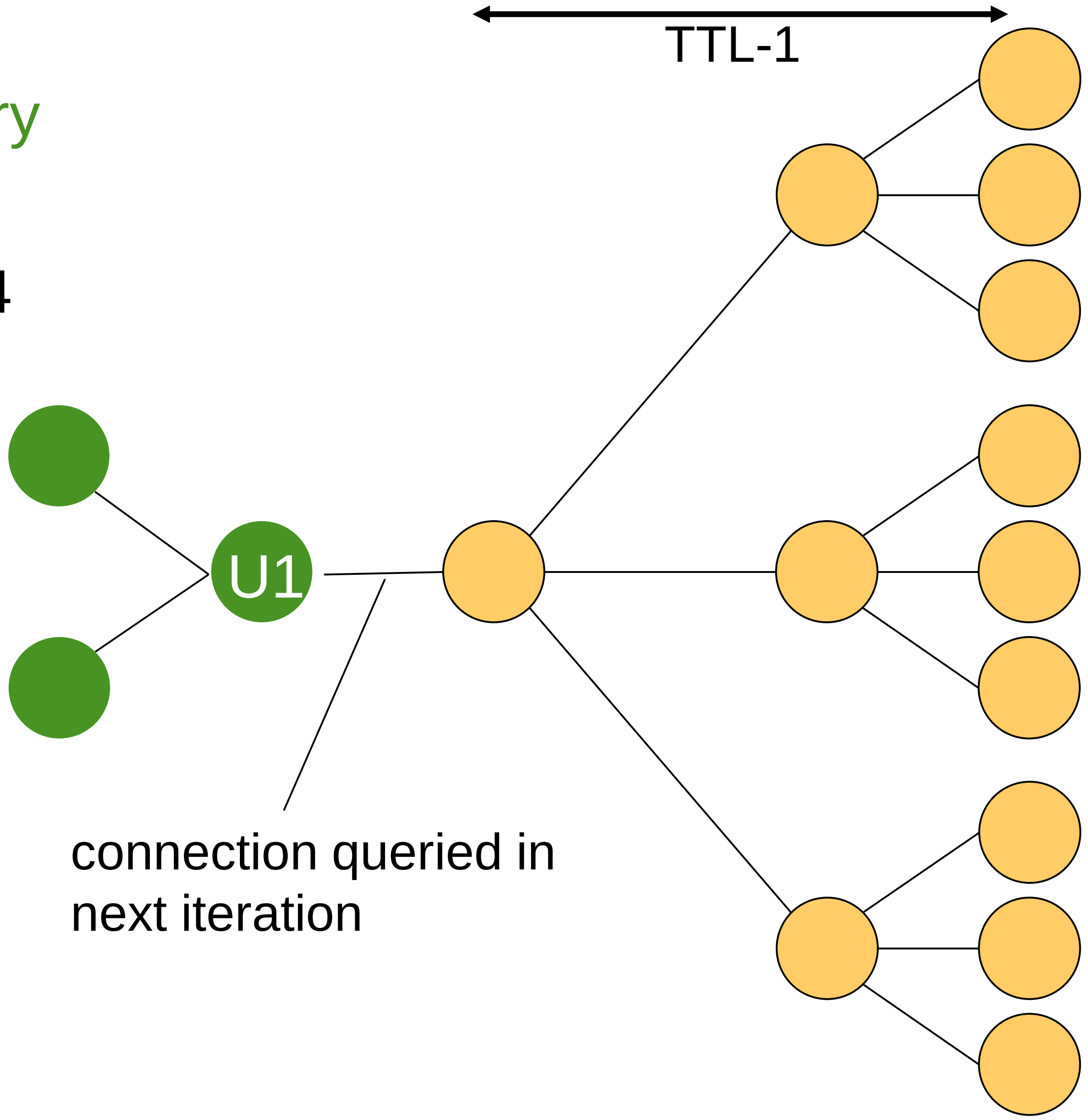
$$\sum_{0 \leq i \leq t-1} (k-1)^i$$

  - Assumptions:
    - All ultra peers have same degree.
    - No clustering.

Gnutella 0.6: Dynamic Querying (7)

Horizon of query

- TTL = 3
- Degree k = 4



Gnutella 0.6: Dynamic Querying (8)

r_h: Expected number of results per ultra peer

- Initial value of r_h is determined based on probe query.
- t (the TTL) is adjusted in each iteration.
- Timeout between successive queries:
  - Deeper queries, i.e., longer paths lead to longer query time.
  - U1 waits for certain timeout to collect results of each iteration.
  - Timeout is chosen according to TTL of query.
  - Typical timeout value: TTL * 2,4 s
  - Within timeout value the majority of results should have arrived at U1.

Dynamic querying is a trade-off between query latency and network load!

When to use an unstructured P2P system?

- Most content is replicated at a fair fraction of hosts:
  - "Most queries are for hay, not needles".
  - Content may be cached or replicated proactively to ensure this.
- 100% recall is not required!
  - User can live with a fraction of all results existing in the network.
- Queries beyond simple id-based queries can be supported more easily than in structured systems.
  - Set of keywords, substrings, etc.

Conclusion (2)

- Important properties of scalable search in unstructured P2P networks:
  - Adaptive query termination
  - Fine-grained network coverage
  - Minimise message duplicates
- Hierarchical super P2P system utilises resources of powerful super peers to improve performance.
- Informed search improves lookup latency through directional “hints”.