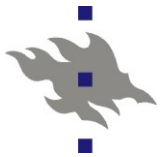
HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Overlay and P2P Networks

Unstructured networks

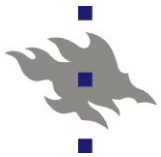
Prof. Sasu Tarkoma

20.1.2014



Contents

- P2P index revisited
- Unstructured networks
  - Gnutella
  - Bloom filters
  - BitTorrent
  - Freenet
- Summary of unstructured networks



P2P Index Revisited

It is crucial to be able to find a data object in the network

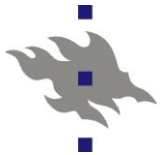
- An index maintains mappings between names and locations

A P2P index can be

- Centralized: single server/farm has the mappings
- Distributed: mappings to a peer are discoverable at a number of other peers
- Local: peer has only local files in the index (and a separate neighbours set)

We already have examples of these

- Centralized: Napster
- Distributed: Skype supernodes, Freenet
- Local: Gnutella V0.4



P2P Indexes Revisited: To Forward?

P2P indexes can also be **forwarding** or **non-forwarding**

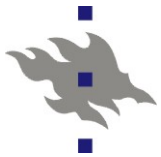
Forwarding indexes (most common) take the request toward the destination based on the indexes of peers that process the request

Non-forwarding indexes take the request directly to the data (typically a single hop)

Examples

Forwarding index: Gnutella V0.7, Freenet

Non-forwarding index: Skype default case with supernodes (relay case is forwarding)



P2P Indexes and Semantics

Most distributed indexes are human-readable and semantic

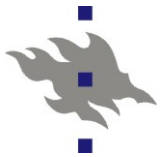
- Keywords, domains, names, …

Unstructured P2P systems support **semantic indexes**

- Can implement various search algorithms (string matching, range queries, …)
- Can support metadata

Semantic-free indexes do not assume semantics but rather have a flat addressing space

- Data centric operation: hash a file to a flat label
- DHT algorithms: efficient routing on flat labels
  - Some node will be responsible for the address space
  - Constraint on where the data is stored
- More difficult to implement string matching or range queries in routing



Gnutella

Gnutella addresses some of Napster's limitations

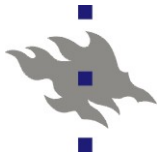
A decentralized P2P system based **flooding the queries**

- Queries are flooded and responses are sent on the **reverse path (with TCP)**
- Downloads directly between peers (HTTP)

Open protocol specification, originally developed by Nullsoft (bought by AOL)

Differs between versions

- 0.4 is the original version (simple flooding)
- 0.7 is more advanced (similar to KaZaa)
  - More structure (hierarchy is good for scalability!)



Gnutella v0.4 protocol messages I

- A peer joining the network needs to discover the address of a peer who is already a member of the network
  - New peer sends GNUTELLA CONNECT message
- A peer then uses PING messages to discover peers and receives PONG messages.
- PONGs include data regarding peers and follow the reverse path of PINGs.

Gnutella v0.4 protocol messages II

- A peer uses the QUERY message to find files, and receives QUERYHIT messages as replies (again on reverse path)
  - Peers forward QUERY messages (flooding)
- The QUERYHIT contains the IP address of the node that can then be used for the file transfer (HTTP)
- PUSH request message can be used to circumvent firewalls (servent sends file to the requesting node after receiving request)
- HTTP Push proxy: proxy sends the push request (V0.7)
  - Requester (HTTP)→ PP (1 hop Gnutella)→ FS (HTTP) → Requester
  - Alleviates problems of reverse path routing

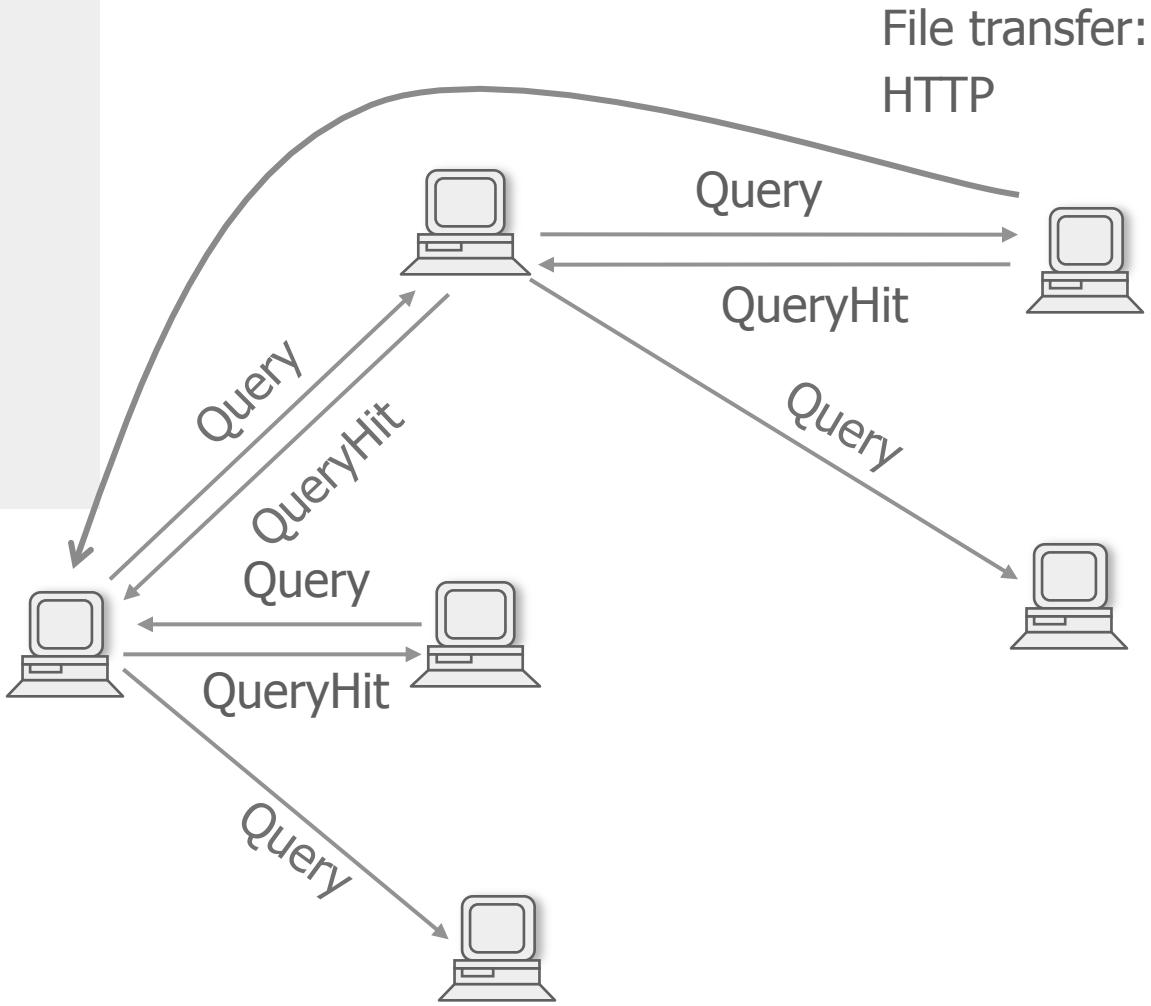
The Gnutella Protocol

Query message sent
- over existing TCP
- connections

Peers forward
- Query message

QueryHit sent over
- reverse path

Scalability:
limited scope
flooding



Gnutella messages

Payload Descriptor:

Ping : does not contain any payload. — 23 B

Pong:

Port	IP	Number of shared files	Size of shared

14 B

Query:

Minimum Speed	Search Criteria

n+1 B

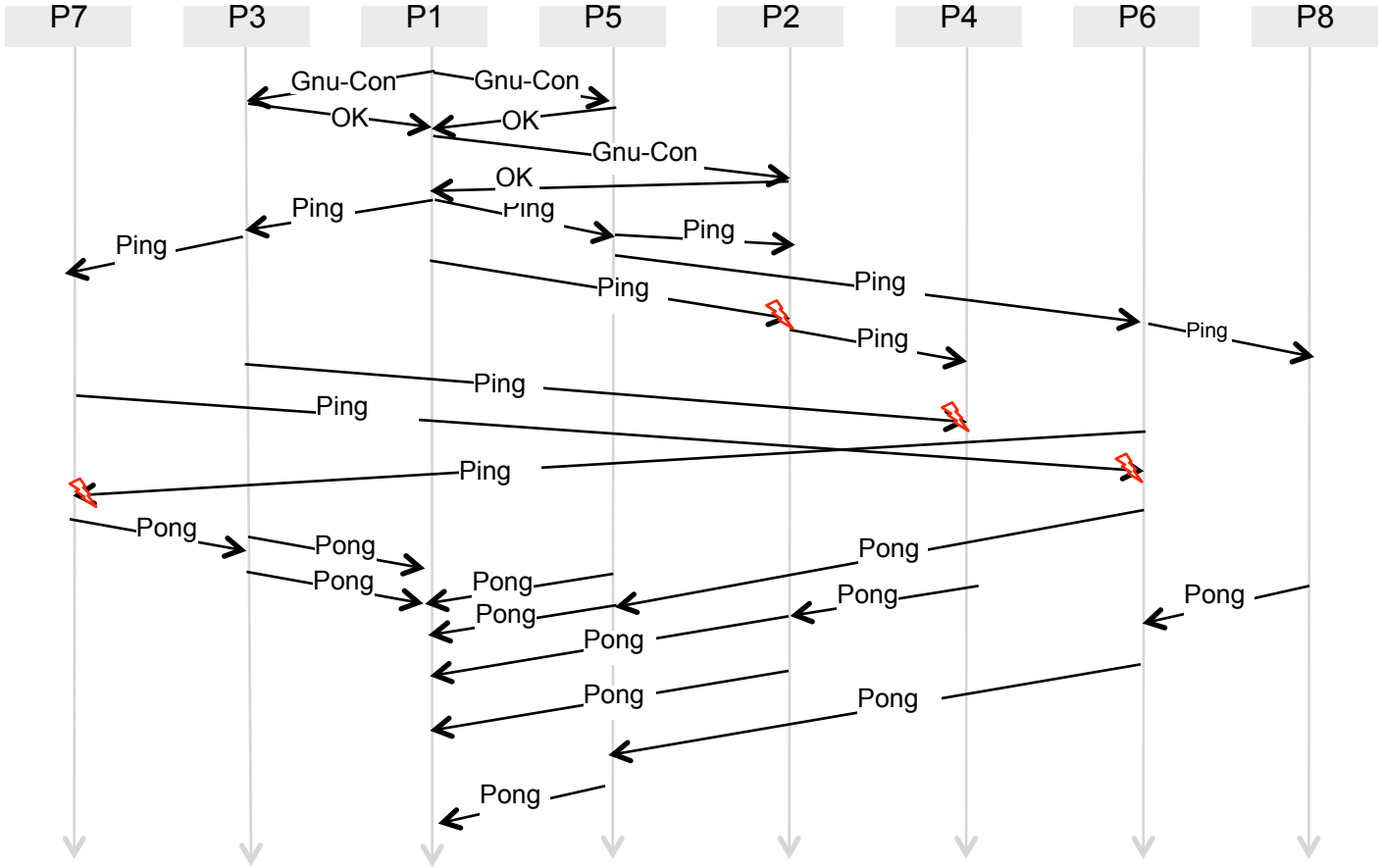
Query_hit:

Number of hits	Port	IP	Speed	Result Set	Node ID

n+16 B

Source: www3.in.tum.de/teaching/ss09/DBSeminar/**P2P**.**ppt**

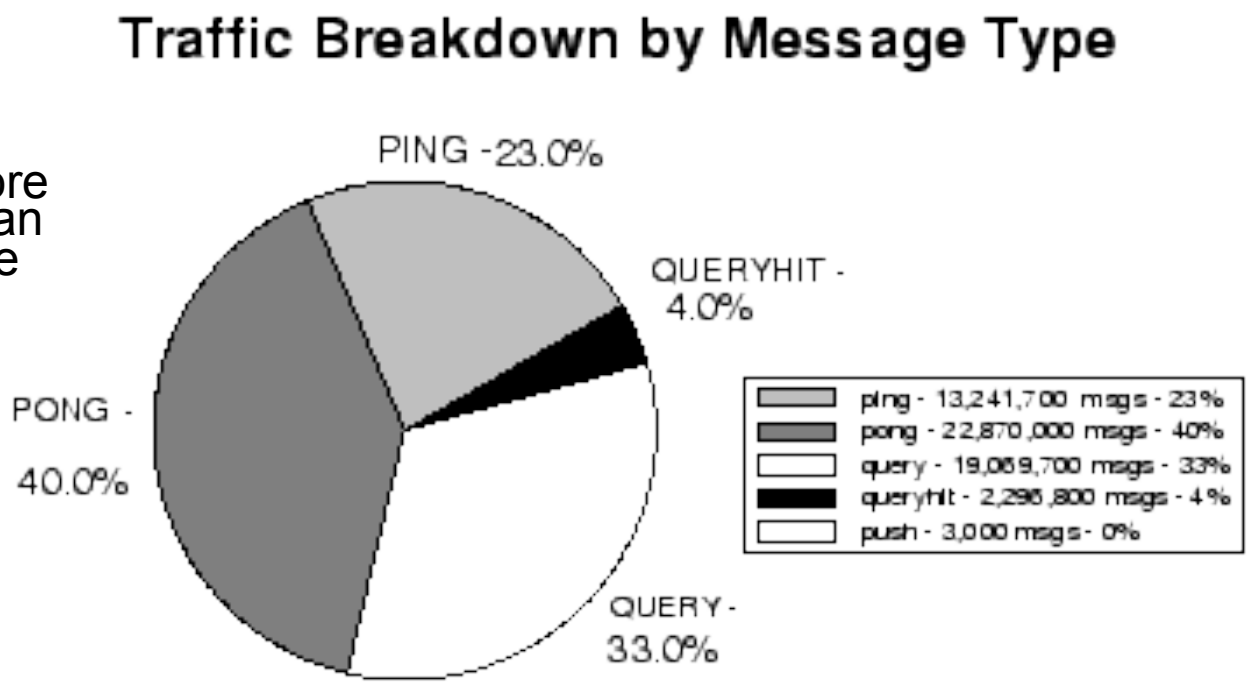
Message propagation



Source: www3.in.tum.de/teaching/ss09/DBSeminar/**P2P**.**ppt**

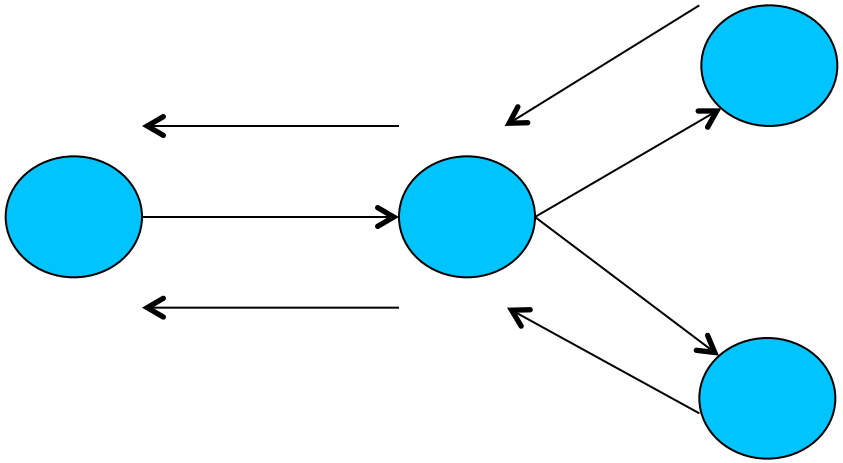
Traffic breakdown

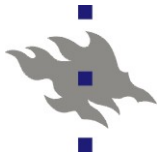
Can be more PONGs than PINGs (see previous diagram)



From "A Quantitative Analysis of the Gnutella Network Traffic"

Pings and Pongs Example





Trees vs graphs

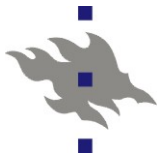
Tree

N nodes, N-1 links

Network with N hosts and M connections, M >= N-1 then (M – (N -1)) loop/redundant connections

These make the network more robust, but increase communications overhead

Loops result in infinite message loops (unless specific loop prevention measures are implemented)



Looping and message processing

Gnutella network is based on a cyclic graph

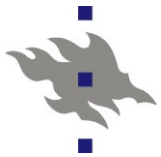
Loops are problematic

Two key solutions:

1. TTL (Time-To-Live): reduces flooding (7 by default)
2. Duplicate detection with unique request identifier

Gnutella uses both (v0.7 is not using flooding anymore so the problem is alleviated)

Even with duplicate detection cannot prevent receiving the same message many times (but can prevent propagation)



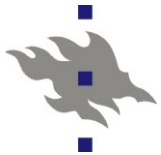
Request messages

Each peer keeps track of all messages it has seen

Can forget about that after some time period

Remember who first sent you a message

If a second copy or subsequent copy of a message arrives,
ignore it

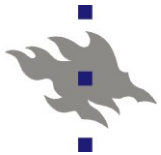


Response messages

Use the same GUID as the message they are in response to

Each peer routes a response msg to the peer from which it first received the original msg

Drop message if did not see original



Problems in original Gnutella reverse path

Peer come and go → routes break

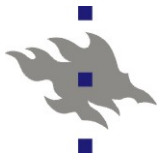
Reverse path requires that the traversed route is used

This means that reverse path may not work

The implementation requires state at the server

Solutions

1. introduce more stable ultra nodes
2. send message toward known content sources → reduce overhead
3. Contact nodes directly!



Review Questions

Q: Does Gnutella guarantee that a file is located?

A: No, the coverage of the network can be tuned with the TTL parameter.

Q: What is the benefit of the local index?

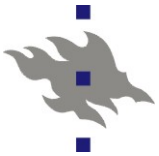
A: It is easy to perform keyword/fine-grained matching.

Q: What is the drawback?

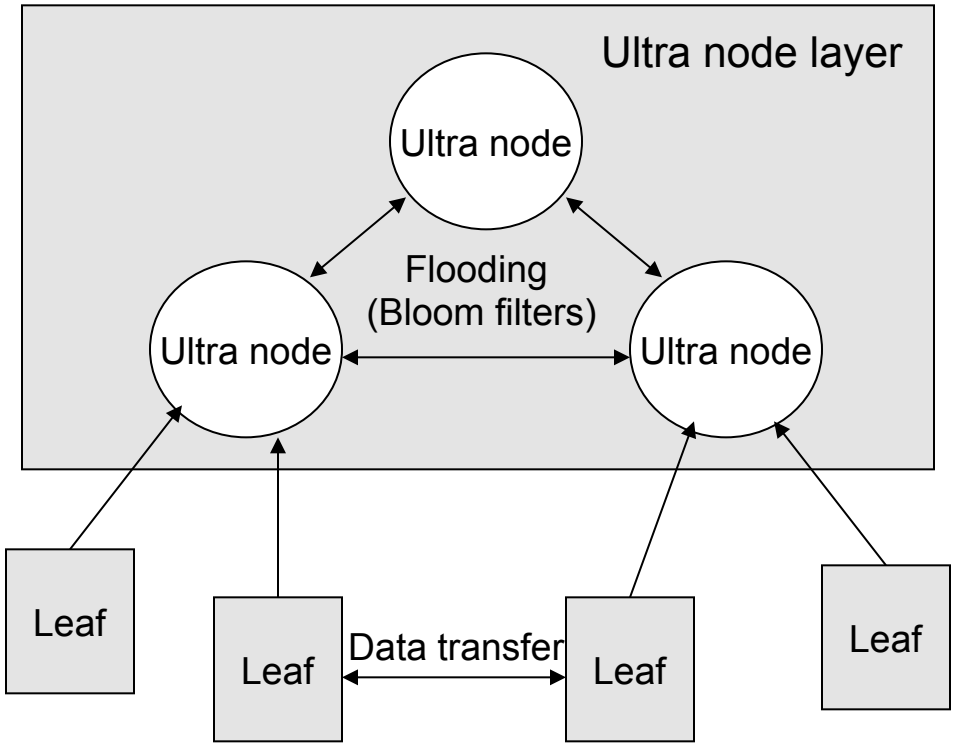
A: Since there is no distributed index, flooding / selected flooding is used to find the files.

Q: What can we do to improve?

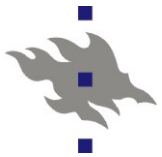
A: Add structure. This allows high-degree nodes to form (hubs) that also makes the system more friendly to the underlying Power Law distribution that has been observed. This results in a significant improvement, but the network is more dependable with the hubs.



The Gnutella v0.7 Architecture



The newer Gnutella uses distributed indexes (at ultra nodes)



Gnutella v0.7 routing

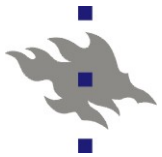
Since version 0.6, Gnutella has been a composite network consisting of leaf nodes and **ultra nodes**. The leaf nodes have a small number of connections to ultra nodes, typically three

The ultra nodes are hubs of connectivity, each being connected to more than 32 other ultra nodes.

When a node with enough processing power joins the network, it becomes an ultra peer and establishes connections with other ultra nodes

This network between the ultra nodes is **flat** and **unstructured**.

These changes attempt to make the Gnutella network reflect the **power-law distributions** found in many natural systems.

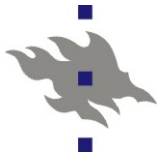


Query Routing Protocol I/III

In Gnutella terminology, the leaf nodes and ultra nodes use the **Query Routing Protocol** to update routing tables, called **Query Routing Table (QRT)**

The QRT consists of a table **hashed keywords** that is sent by a leaf node to its ultra nodes

Ultra nodes merge the available QRT structures that they have received from the leaf nodes, and exchange these merged tables with their neighbouring ultra nodes



Query Routing Protocol II/III

Query routing is performed by hashing the search words and then testing whether or not the resulting hash value is present in the QRT

Ultrapeer forwards query to top-level connections and waits for responses

Query is flooded outward until the TTL expires

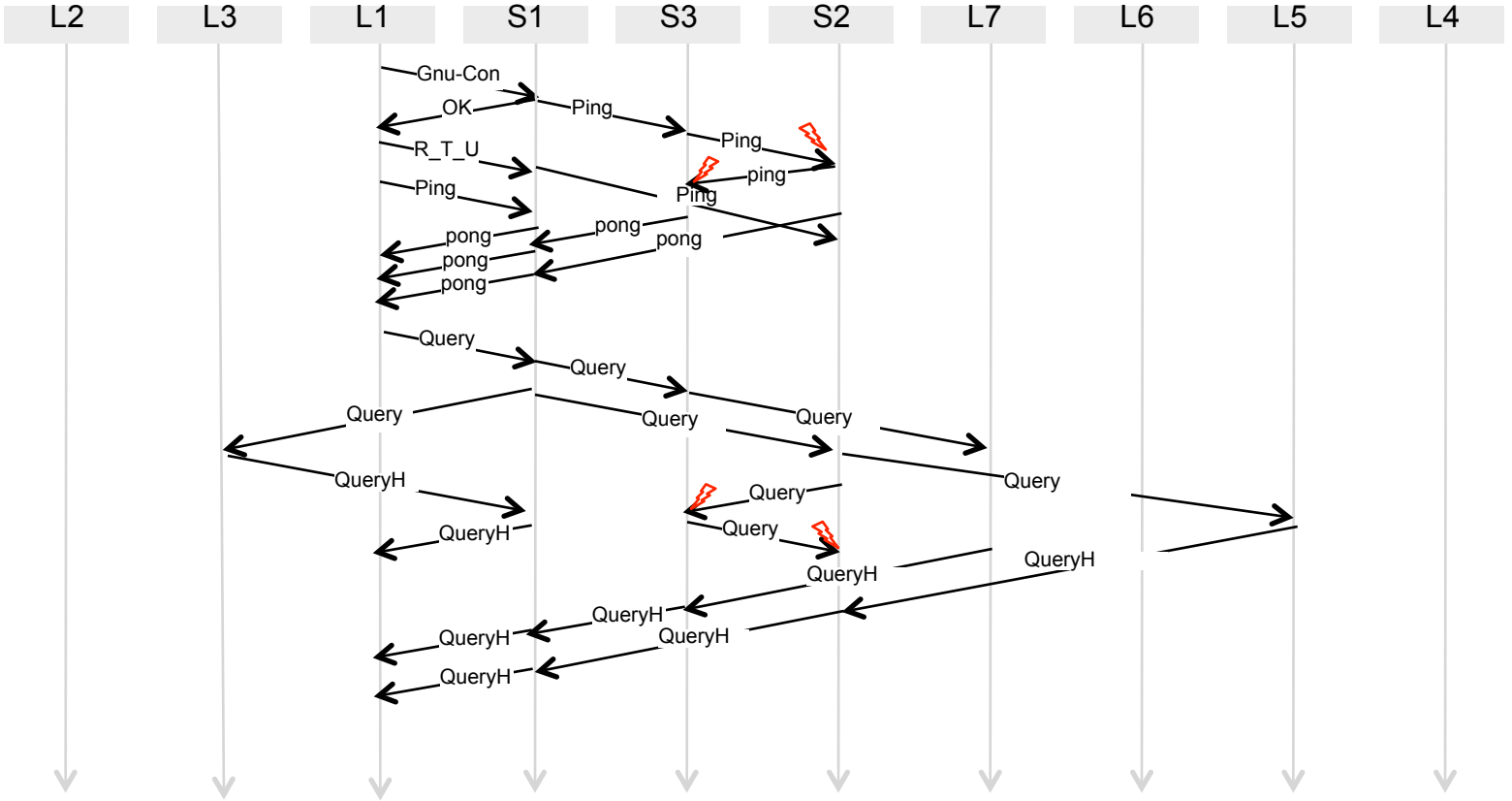
Query Routing Protocol III: Tuning the TTL

The ultrapeer then waits for the results, and determines how **rare matches are** (the ratio between the number of results and the estimated number of visited peers)

If matches are rare, the query is sent through more connections with a relatively high TTL

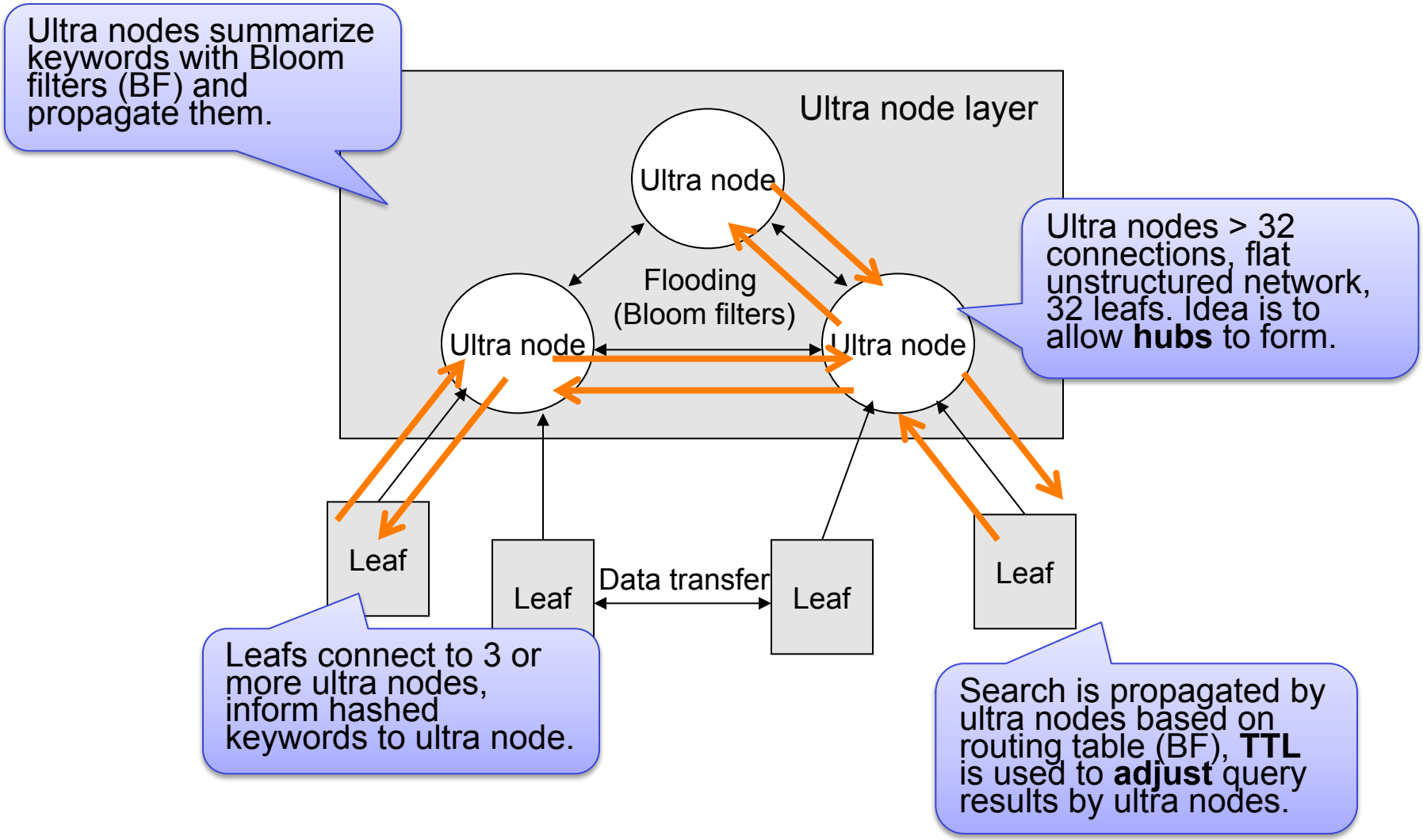
If matches are more common but not sufficient, the query is sent down a few more connections with a low TTL

Message Propagation



Source: www3.in.tum.de/teaching/ss09/DBSeminar/**P2P**.**ppt**

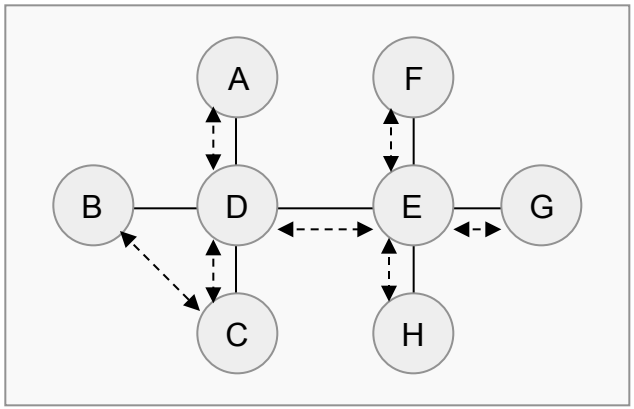
The new Gnutella Architecture



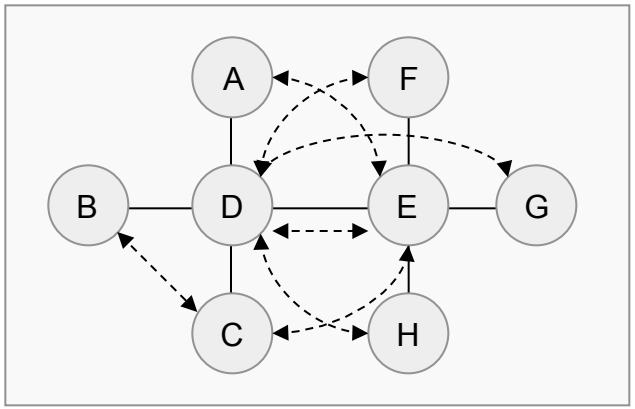
Mapping the Gnutella Network

Map the network by crawling or monitoring hubs

Example: Gnutella v0.4 random topology has problems



Perfect mapping for message from A. Link D-E is traversed only once.

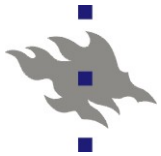


Inefficient mapping that results in link D-E being traversed six times

Overlay networks can result in really bad application layer routing configurations unless the underlay is taken into account!

Hubs help here if they are chosen wisely.

Clustering can result in 3-5 orders of magnitude better performance than Gnutella v0.4



Improving Gnutella Search I/II

Search has a tradeoff between network traffic, peer load, and probability of a query hit

Three techniques:

Flooding: not scalable and results in a lot of traffic

Ring: Have a fixed TTL for the search. This was found to be problematic: how to set the TTL?

Expanding ring (iterative deepening): successively larger TTL counter until there is a match

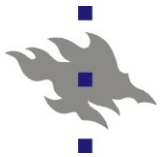
These increase network load with duplicated query messages.

Alternative technique: **random walks**

Query wanders about the network: reduces network load but increases search latency

Random k-walkers: replicate random-walks

Also a number of policy-based and probabilistic techniques

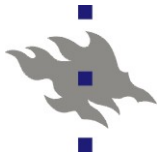


Improving Gnutella Search II

- **Selective** flooding can be combined with spanning trees, random walks, etc. Good for bootstrapping search.
- GIA by Y. Chawathe et al. (SIGCOMM 2003) outperforms Gnutella v0.4 by 3-5 orders of magnitude

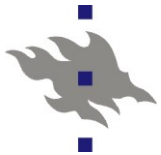
- Design principles
  - Explicitly account for **node heterogeneity**
  - Query load **proportional** to node *capacity*

  - Make **high-capacity nodes** easily reachable
    - Dynamic topology adaptation converts them into high-degree nodes
  - Make high-capacity nodes have more answers
  - Biased random walks and overload avoidance
- These results influenced the Gnutella V0.7 design



	Gnutella v0.4	Gnutella v0.7
Decentralization	Flat topology (random graph), equal peers	Random graph with two tiers. Two kinds of nodes, regular and ulta nodes. Ultra nodes are connectivity hubs
Foundation	Flooding mechanism	Selective flooding using the super nodes
Routing function	Flooding mechanism	Selective flooding mechanism
Routing performance	Search until Time-To-Live expires, no guarantee to locate data	Search until Time-To-Live expires, second tier improves efficiency, no guarantee to locate data
Routing state	Constant (reverse path state, max rate and TTL determine max state)	Constant (regular to ultra, ultra to ultra). Ultra nodes have to manage leaf node state.
Reliability	Performance degrades when the number of peer grows. No central point.	Performance degrades when the number of peer grows. Hubs are central points that can be taken out.

A Short Primer on Bloom Filters



Bloom filters in Gnutella v0.7

Bloom filters are probabilistic structures used to store dictionaries

A bit-vector that supports constant time querying of keywords

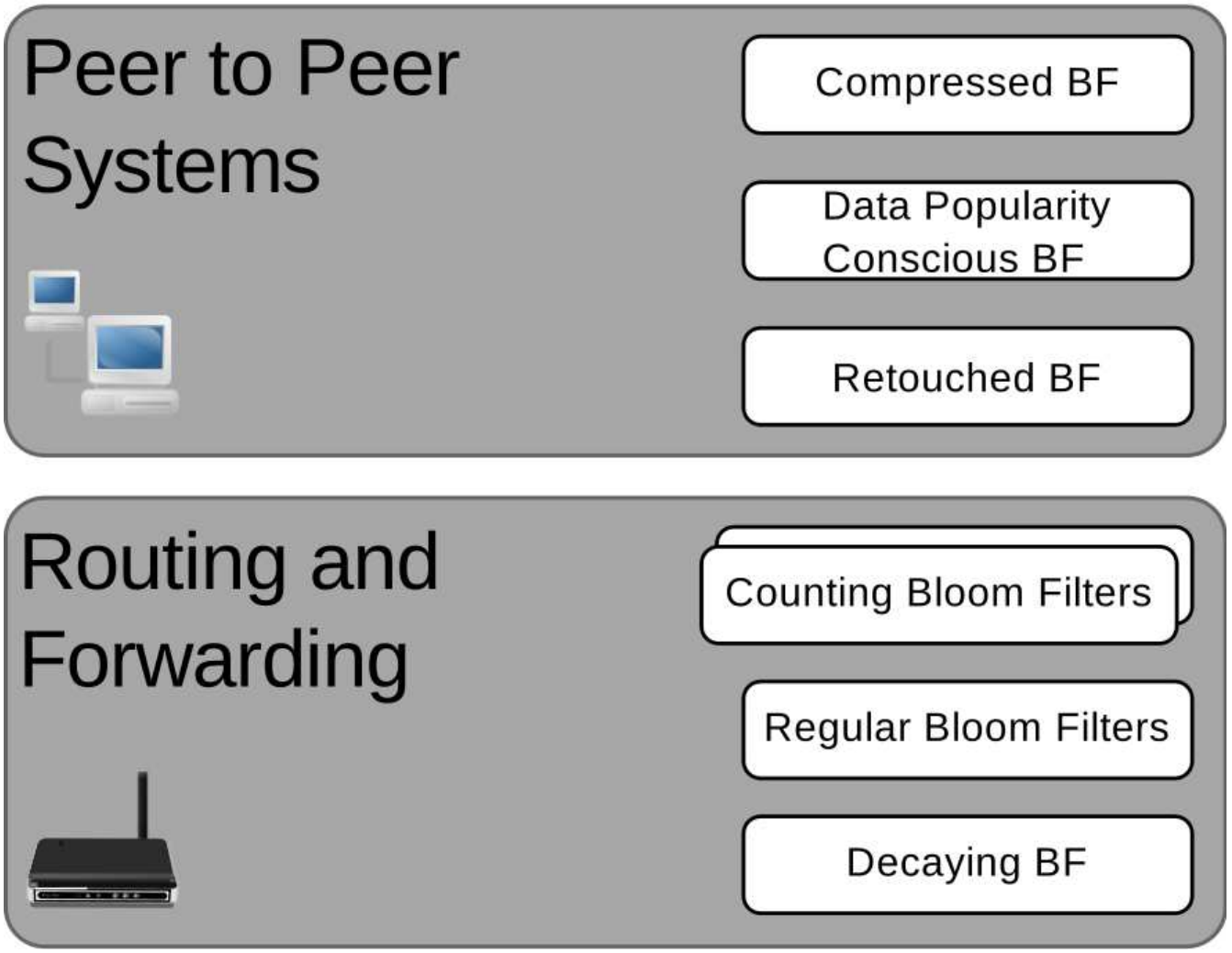
Easy to merge two filters

Many variants

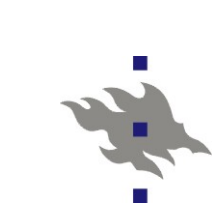
If space is at premium

	Decrease	Increase
Number of hash functions (k)	Less computation Higher false positive rate	More computation Lower false positive rate
Size of filter (m)	Smaller space requirements Higher false positive rate	More space is needed Lower false positive rate
Number of elements in the inserted set (n)	Lower false positive rate	Higher false positive rate

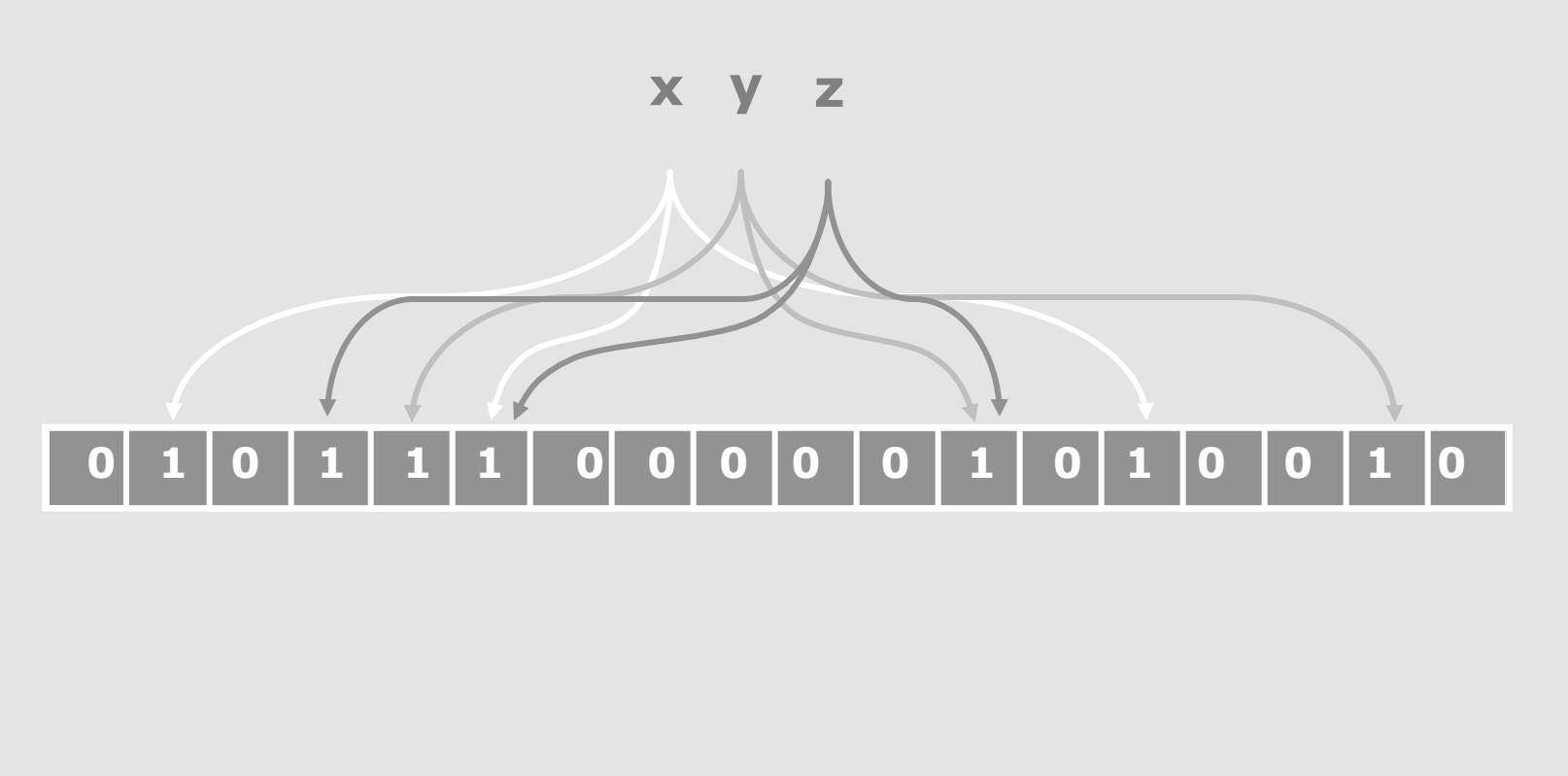
Bloom Filters

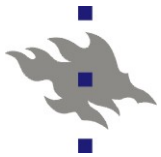


Source: Tarkoma et al. Theory and Practice of Bloom Filters for distributed Systems. 2013.



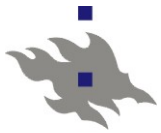
Example Bloom filter





```
Data: x is the object key to insert into the Bloom filter.
Function: insert(x)
for j : 1...k do
    /* Loop all hash functions k                    */
    i ← h_j(x);
    if B_i == 0 then
        /* Bloom filter had zero bit at
        position i                                   */
        B_i ← 1;
    end
end
```

Algorithm 1: Pseudocode for Bloom filter insertion



Data: x is the object key for which membership is tested.
Function: *ismember(x)* returns true or false to the membership test

$m \leftarrow 1$;
$j \leftarrow 1$;
while $m == 1$ *and* $j \leq k$ **do**
    $i \leftarrow h_j(x)$;
    if $B_i == 0$ **then**
        $m \leftarrow 0$;
    end
    $j \leftarrow j + 1$;
end
return m;

Algorithm 2: Pseudocode for Bloom member test

BF False positive probability is given by:

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1-e^{-kn/m}\right)^{k}.$$

All bit positions are set to one for k hash functions

Size of filter given optimal number of hash functions:

$$m = -\frac{n \ln p}{(\ln 2)^2}.$$

Details in the survey paper available on course page.

BF False positive probability is given by:

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k.$$

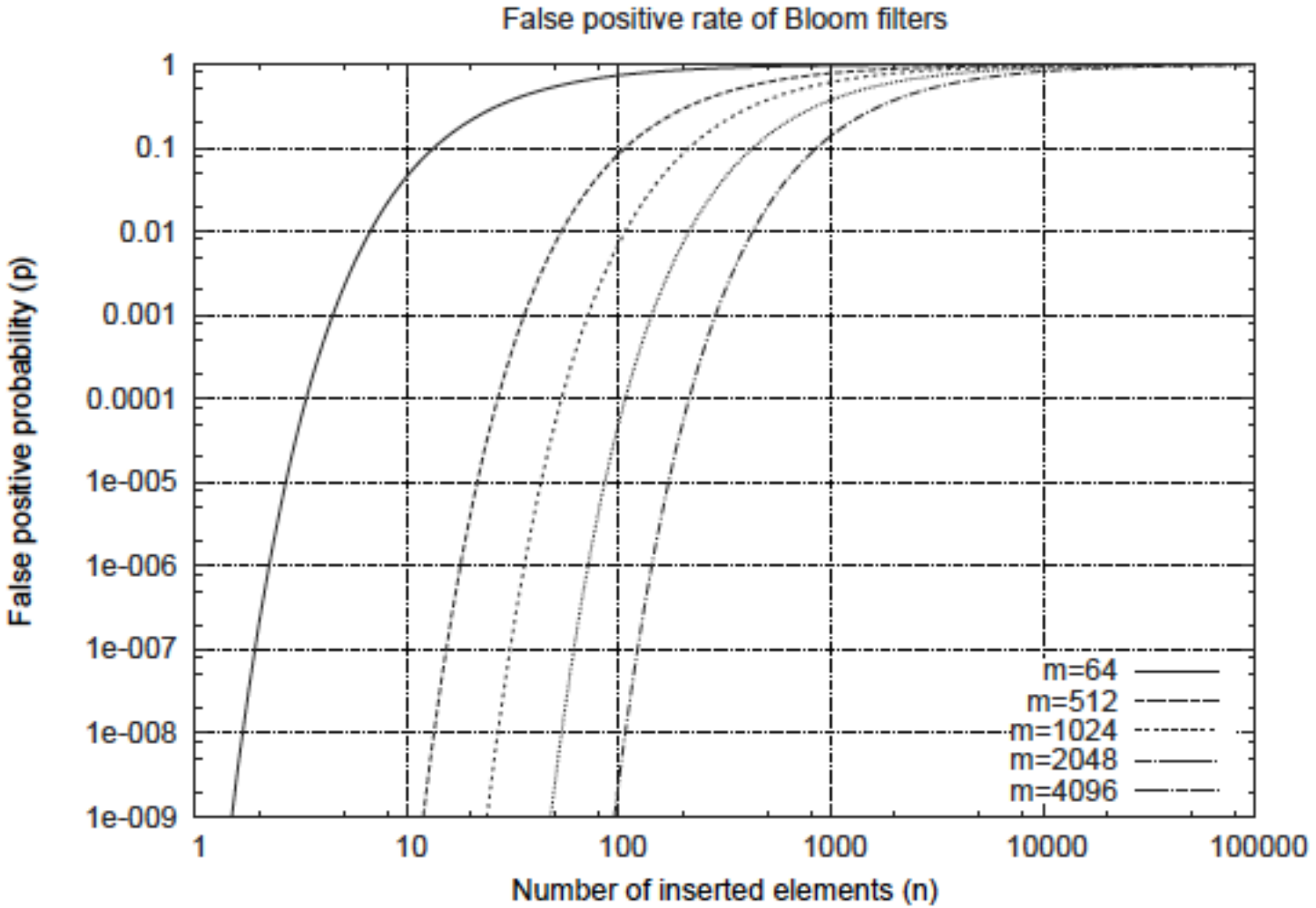
Optimal number of hash functions k:

$$k_{opt} = \frac{m}{n} \ln 2 \approx \frac{9m}{13n}.$$

Size of filter given optimal number of hash functions:

$$m = -\frac{n \ln p}{(\ln 2)^2}.$$

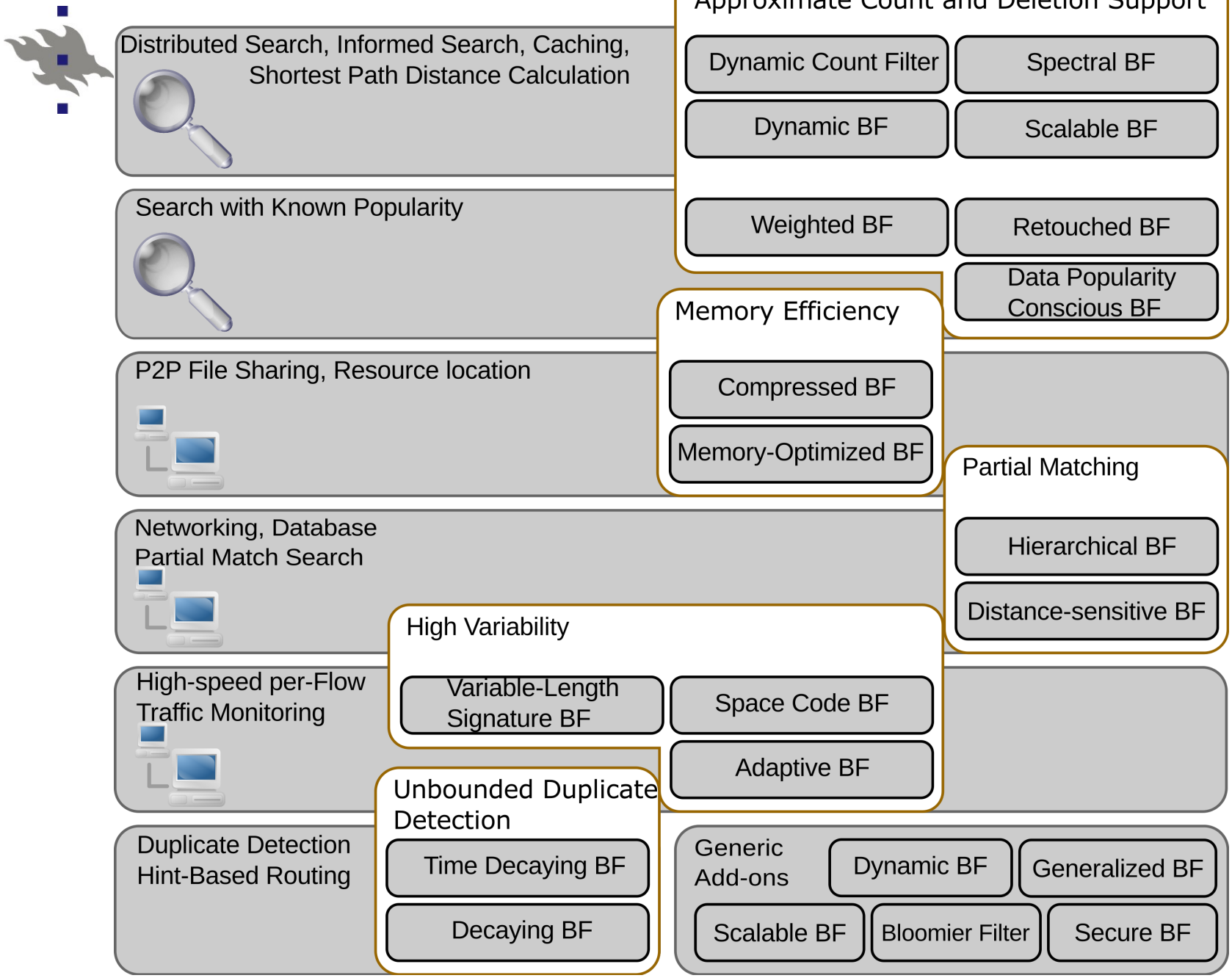
Details in the survey paper available on course page.



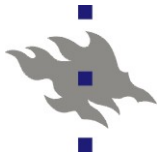
If false positive rate is fixed, the filter size grows linearly with inserted elements.

Filter	Key feature	C	D	P	FN	Output
Standard Bloom filter	Is element x in set S?	N	N	N	N	Boolean
Adaptive Bloom filter	Frequency by increasing number of hash functions	Y	N	N	N	Boolean
Bloomier filter	Frequency and function value	Y	N	N	N	Freq., $f(x)$
Compressed Bloom filter	Compress filter for transmission	N	N	N	N	Boolean
Counting Bloom filter	Element frequency queries and deletion	Y	Y	N	M	Boolean or freq.
Decaying Bloom filter	Time-window	Y	Y	N	N	Boolean
Deletable Bloom filter	Probabilistic element removal	N	Y	N	N	Boolean
Distance-sensitive Bloom filters	Is x close to an item in S?	N	N	N	Y	Boolean
Dynamic Bloom filter	Dynamic growth of the filter	Y	Y	N	N	Boolean
Filter Bank	Mapping to elements and sets	Y	Y	M	N	x, set, freq.
Generalized Bloom filter	Two set of hash functions to code x with 1s and 0s	N	N	N	Y	Boolean
Hierarchical Bloom filter	String matching	N	N	N	N	Boolean
Memory-optimized Bloom filter	Multiple-choice single hash function	N	N	N	N	Boolean
Popularity conscious Bloom filter	Popularity-awareness with off-line tuning	N	N	Y	N	Boolean
Retouched Bloom filter	Allow some false negatives for better false positive rate	N	N	N	Y	Boolean
Scalable Bloom filter	Dynamic growth of the filter	N	N	N	N	Boolean
Secure Bloom filters	Privacy-preserving cryptographic filters	N	N	N	N	Boolean
Space Code Bloom filter	Frequency queries	Y	N	M	N	Frequency
Spectral Bloom filter	Element frequency queries	Y	Y	N	M	Frequency
Split Bloom filter	Set cardinality optimized multi-BF construct	N	N	N	N	Boolean
Stable Bloom filter	Has element x been seen before?	N	Y	N	Y	Boolean
Variable-length Signature filter	Popularity-aware with on-line tuning	Y	Y	Y	Y	Boolean
Weighted Bloom filter	Assign more bits to popular elements	N	N	Y	N	Boolean

Source: Tarkoma et al. Theory and Practice of Bloom Filters for distributed Systems. 2013.



Source: Tarkoma et al. Theory and Practice of Bloom Filters for distributed Systems. 2013.



Worked Example

- Gnutella uses Bloom filters to store and disseminate keyword indexes
- 1-hop replication in the flat ultranode layer, much improved design over flooding
- An ultrapeer maintains about 30 leafs and thus 30 Bloom filters, one for each leaf
- One leaf has about 1000 keywords in our example
- Assuming false positive rate of 0.1, for 1000 keywords we need 4793 bits. For 30 000 keywords we need 143 776 bits.
- There is overlap (some keywords are popular)!
- Gnutella uses 2**16 (65536) bits that is sufficient even when aggregating leaf BFs
- Experiments report ultrapeer BF 65% full and leaf BF 3% full
- Today having hundreds of KBs in the BF is not a problem, Gnutella design is old and today's networks are much faster

Linear growth if p is fixed!

$$m = -\frac{n \ln p}{(\ln 2)^2}.$$