Overlay and P2P Networks

Unstructured networks

Prof. Sasu Tarkoma

19.1.2015
Contents

• Unstructured networks
  – Last week
    – Napster
    – Skype
  – This week:
    – Gnutella
    – BitTorrent
P2P Index

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
Local: Gnutella V0.4 and hybrid Gnutella V0.7
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

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 label. 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 on **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.

Flooding is not efficient!
Improving capability: limited scope flooding.

File transfer: HTTP
Gnutella Messages

Gnutella messages have two parts
- Descriptor header
- Payload header

Descriptor header consists of:
- Descriptor ID (16-byte string)
- Payload Descriptor code (type of message)
- TTL (this is decremented by each servent to 0 when forwarding stops)
- Hops (the number of hops the descriptor has travelled)
- Payload length
# Gnutella messages

## Payload Descriptor:

**Ping:** does not contain any payload.

**Pong:**

<table>
<thead>
<tr>
<th>Port</th>
<th>IP</th>
<th>Number of shared files</th>
<th>Size of shared</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>14 B</td>
</tr>
</tbody>
</table>

**Query:**

<table>
<thead>
<tr>
<th>Minimum Speed</th>
<th>Search Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n+1 B</td>
</tr>
</tbody>
</table>

**Query_hit:**

<table>
<thead>
<tr>
<th>Number of hits</th>
<th>Port</th>
<th>IP</th>
<th>Speed</th>
<th>Result Set</th>
<th>Node ID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n+16 B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
Broadcasting in Gnutella network

Three important metrics for the search cost:
- Number of nodes
- TTL value
- Number of neighbors

Search cost is demonstrated by the Caley tree:
- a rooted tree in which nodes are organized around the root
- each node has $z$ neighbors
- an infinite connected cycle-free graph

Our case of forwarding to constant $z$ neighbors has the cost of:
$$z + z^2 + z^3 + \ldots$$

Figure source: [http://en.wikipedia.org/wiki/Bethe_lattice#mediaviewer/File:Bethe_lattice.png](http://en.wikipedia.org/wiki/Bethe_lattice#mediaviewer/File:Bethe_lattice.png)
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

Peers 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 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.
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
The new Gnutella Architecture

Ultra nodes summarize keywords with Bloom filters (BF) and propagate them.

Ultra nodes > 32 connections, flat unstructured network, 32 leafs. Idea is to allow hubs to form.

Leafs connect to 3 or more ultra nodes, inform hashed keywords to ultra node.

Search is propagated by ultra nodes based on routing table (BF), TTL is used to adjust query results by ultra nodes.
Mapping the Gnutella Network

Map the network by crawling or monitoring hubs

Example: Gnutella v0.4 random topology has problems

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

<table>
<thead>
<tr>
<th></th>
<th>Decrease</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of hash functions (k)</td>
<td>Less computation</td>
<td>More computation</td>
</tr>
<tr>
<td></td>
<td>Higher false positive rate</td>
<td>Lower false positive rate</td>
</tr>
<tr>
<td>Size of filter (m)</td>
<td>Smaller space requirements</td>
<td>More space is needed</td>
</tr>
<tr>
<td></td>
<td>Higher false positive rate</td>
<td>Lower false positive rate</td>
</tr>
<tr>
<td>Number of elements in the inserted set (n)</td>
<td>Lower false positive rate</td>
<td>Higher false positive rate</td>
</tr>
</tbody>
</table>
Bloom Filters

Fig. 9. Bloom filter variants grouped by usage scenarios.

Since there is no Bloom filter that fits all, one key question that application designers should ask is whether false negatives are tolerable or not. Relaxing this constraint can help drastically in reducing the overall false positive rate (cf. retouched Bloom filters [50]), but raises also the question whether the Bloom filter is the right data structure choice despite alternative designs specific to the application domain (cf. [61]), approximate dictionary-inspired approaches [6], [35], cache-efficient variants (blocked Bloom filter) and Golomb coding implementations as proposed by Putze et al [62], space-efficient versions of cuckoo hashing [63], and more complex but space-optimal alternatives [5], [6].

Each variant or replacement introduces a specific trade-off involving execution time, space efficiency, and so on. Ultimately, which probabilistic data structure is best suited depends a lot on the application specifics. Indeed, the variations of the standard Bloom filter discussed in this Section are commonly the result of specific requirements of network and distributed system applications, a variety of which we present in the following survey section.

IV. BLOOM FILTERS IN DISTRIBUTED COMPUTING

We have surveyed techniques for probabilistic representation of sets and functions. The applications of these structures are manyfold, and they are widely used in various networking systems, such as Web proxies and caches, database servers, and routers. We focus on the following key usage scenarios:

- Caching for Web servers and storage servers.
- Supporting processing in P2P networks, in which probabilistic structures can be used for summarizing content and caching [28], [64].
- Packet routing and forwarding, in which Bloom filters and variants have important roles in flow detection and classification.
- Monitoring and measurement. Probabilistic techniques can be used to store and process measurement data summaries in routers and other network entities.
- Supporting security operations, such as flow admission and intrusion detection.

Figure 9 shows an overview of Bloom filter variants that can be used in the usage scenarios that this section focuses on. For more detail, see Figure 15 at the end of this article.

A. Caching

Bloom filters have been applied extensively to caching in distributed environments. To take an early example, Fan, Cao, Almeida, and Broder proposed the Summary Cache [27], [28] system, which uses Bloom filters for the distribution of Web cache information. The system consists of cooperative proxies that store and exchange summary cache data structures, essentially Bloom filters. When a local cache miss happens, the proxy in question will try to find out if another proxy has a copy of the Web resource using the summary cache. If another proxy has a copy, then the request is forwarded there.

In order for distributed proxy-based caching to work well, the proxies need to have a way to compactly summarize available content. In the Summary Cache system, proxies periodically transfer the Bloom filters that represent the cache contents (URL lists). Figure 10 illustrates the use of a Bloom filter-based summary cache at a proxy. The summary cache is consulted and used to find nearest servers or other proxies with the requested content.

Dynamic content poses a challenge for caching content and keeping the summary indexes up to date. Within a single proxy, a Bloom filter representing the local content cache needs to be recreated when the content changes. This can be seen to be inefficient and as a solution the Summary Cache uses counting Bloom filters for the maintenance of their local cache contents, and then based on the updates a regular Bloom filter is broadcast to other proxies.

The summary cache-based technique is used in the popular Squid Web Proxy Cache. Squid uses Bloom filters for so-called cache digests. The system uses a 128-bit MD5 hash of the key, a combination of the URL and the HTTP method, and splits the hash into four equal chunks. Each chunk modulo the digest size is used as the value for one of the Bloom filter hash functions. Squid does not support deletions from the digest and thus the digest must be periodically rebuilt to remove stale information.

Bloom filters have been applied extensively in distributed storage to minimize disk lookups. As an example, we consider...

Example Bloom filter
Data: $x$ is the object key to insert into the Bloom filter.

Function: $\text{insert}(x)$

for $j : 1 \ldots k$ do

   /* Loop all hash functions $k$ */
   $i \leftarrow h_j(x)$;

   if $B_i == 0$ then

      /* Bloom filter had zero bit at position $i$ */
      $B_i \leftarrow 1$;

   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

\[
m \leftarrow 1; \\
j \leftarrow 1; \\
\textbf{while } m == 1 \textbf{ and } j \leq k \textbf{ do} \\
\hspace{1em} i \leftarrow h_j(x); \\
\hspace{1em} \textbf{if } B_i == 0 \textbf{ then} \\
\hspace{2em} m \leftarrow 0; \\
\hspace{1em} \textbf{end} \\
\hspace{1em} j \leftarrow j + 1; \\
\textbf{end} \\
\textbf{return } m;
\]

Algorithm 2: Pseudocode for Bloom member test
BF False positive probability is given by:

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

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.
Bloom filters are a data structure designed to efficiently determine if an element is in a set, with a trade-off between memory usage and accuracy. They are widely used in distributed systems due to their compact size and low memory footprint.

### Table II: Key Features of Bloom Filter Variants

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

In [130], Bloom filters are used to represent and query ranges of multi-dimensional data. Range queries are handled by segmenting the attribute range into separate Bloom filters that represent membership in that segment.

V. SUMMARY

Bloom filters are a general aid for network processing and improving the performance and scalability of distributed systems. In Figure 15, the Bloom filter variants introduced in this paper are categorized by application domain and supported features. The figure aims to help domain experts select an appropriate Bloom filter based on their application. An expert need only find their domain on the left side and pick a Bloom filter on its right. Each rectangular bubble represents a Bloom filter variant. Variants that support a certain feature are found inside a highlighted area labeled with the name of that feature.

- **Approximate Count and Deletion Support**
  - Dynamic Count Filter
  - Dynamic BF
  - Scalable BF
  - Spectral BF

- **Memory Efficiency**
  - Compressed BF
  - Memory-Optimized BF
  - Data Popularity Conscious BF

- **Partial Matching**
  - Hierarchical BF
  - Distance-sensitive BF

- **High Variability**
  - Variable-Length Signature BF
  - Space Code BF
  - Adaptive BF

- **Unbounded Duplicate Detection**
  - Time Decaying BF
  - Decaying BF

- **High-speed per-Flow Traffic Monitoring**

- **Search with Known Popularity**

- **P2P File Sharing, Resource location**

- **Networking, Database Partial Match Search**

- **Duplicate Detection Hint-Based Routing**

- **Generic Add-ons**
  - Dynamic BF
  - Generalized BF
  - Scalable BF
  - Bloomier Filter
  - Secure BF

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 leaves 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

\[ m = - \frac{n \ln p}{\left(\ln 2\right)^2}. \]