Bloom Filter-based Stateless Multicast

Éva Hosszu
hosszu@tmit.bme.hu
Outline

1. Multicast in publish/subscribe networks
   1. Pub/sub network architecture
2. Bloom filter basics
   1. What is a Bloom filter?
   2. False positive probability
3. Stateless Forwarding on Bloomed link identifiers
   1. Bloom-filter based multicast forwarding method
   2. Limitations
4. Concluding remarks
Stateless Multicast

- **Multicast**: one-to-many communication
  - Delivery of a message or information to a group of destination computers simultaneously in a single transmission from the source.
  - Unicast → Multicast → Broadcast
  - Send an e-mail to a mailing list
  - RSS feed

- **Stateless**: each request is treated independently
  - Unrelated to previous requests
  - Independent pairs of requests and responses
  - E.g. IP, HTTP
  - as opposed to a stateful FTP server
Publish/subscribe network architecture

- Multicast forwarding fabric
- Offers decoupling in time, space and desynchronization
- Recursive structure
- Each higher layer utilizes the functionalities of the lower layers
- Bottom: forwarding fabric
Control plane functionalities

- **Topology system**
  - Creates a distributed awareness of the structure of the network

- **On top of it: Rendezvous system**
  - Handles the matching between publishers and subscribers
  - Active subscriber → requests the topology to construct a forwarding tree & to provide the publisher with suitable forwarding information
Data plane functionalities

- Forwarding functionality
- Traditional transport functions
  - Error detection
  - Traffic scheduling
- New network functions
  - Opportunistic caching
  - Lateral error correction

- Data and control plane functions work in concert
  - Organized into an unlayered architecture
  - Utilize each other in a component wheel
Outline

1. Multicast in publish/subscribe networks
   1. Pub/sub network architecture

2. Bloom filter basics
   1. What is a Bloom filter?
   2. False positive probability

3. Forwarding on Bloomed link identifiers
   1. Bloom-filter based multicast forwarding method
   2. Limitations

4. Concluding remarks
Bloom filter

- Data structure designed to represent a set to support membership queries
  - Simple
  - Space-efficient
  - Randomized
- Given Universe \( U \); a set \( S \) in \( U \): is \( x \) in \( S \)?
  - May return a false positive
  - Collaborating in overlay and peer-to-peer networks
  - Resource routing
  - Packet routing
  - Google BigTable
- \( m \)-bit long binary array with some bits set to 1
  - Supported operations: Insert, Query
Bloom Filter Original: Hyphenation

- Program for automatic hyphenation
- 90% of English words can be hyphenated using a few simple rules
- 10% require a lookup
- Entire dictionary is too large to be kept in core memory
- By allowing errors: hash area can be made sufficiently small
  - Bloom filter of the 10% fits in core memory
- False positive: unrequired lookup
  - Rare occurrence
How a Bloom filter works: Insert

- Universe U of elements, 1..N
- S ⊆ U of n elements, x₁, x₂, ..., xₙ
- Start: m bits all set to 0
- Choose k hash functions
  - Evenly distributed among m bits
  - Implementation: divide into k subsets
- Hash each element in S k times
- Set the corresponding bits to 1
How a Bloom filter works: Query

- Given a Bloom filter
  - \( m \) bits, some of them are set to 1, rest are 0
- Query(\( x \)):
  - Hash \( x \) with the \( k \) hash functions
  - Check if the corresponding bits are 1 in the filter
    - If yes: \( x \) is probably in the set (may be a false positive)
    - If no: \( x \) is definitely not in the set
Bloom filter example

- **Start:**

- **Insert:**

- **Query:**

Example: Add 18

18

Definitely not there.
Example: Add 25

 Definitely not there.
Example: Add 6

Definitely not there.
Example: Add 14

Definitely not there.
Query 18: YES
Query 5: NO

Definitely not there.
Query 20: NO

Definitely not there.
Query 23: YES $\rightarrow$ false positive
Are the queries always right?

- False positive may occur
- **False positive:** query(x) returns positive answer, even though x is not in S
- **False positive probability:**
  - k hash functions
  - m bits long array
  - After inserting n elements, a specific bit is still 0:

\[
p' = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}
\]
False positive probability

- Let \( \rho \) be the proportion of 0 bits after all elements are inserted in the filter
- Expected value is \( \mathbb{E}(\rho) = \rho' \)
- Conditioned on \( \rho \), the probability of a false positive is:
  \[
  (1 - \rho)^k \approx (1 - \rho')^k
  \]
- That is,
  \[
  f' = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k
  \]
Optimal number of hash functions

- Given filter-length $m$ and the number of elements $n$, one can optimize the number of hash functions.
- Find $k$, such that the false positive probability $f'$ is minimal.
- Derivation yields:
  \[ k = \ln 2 \cdot \left( \frac{m}{n} \right) \]

Example:
- Let $m = 256$, $n = 25$
- $k = \ln 2 \cdot (256/25) \approx 7.09 \approx 7$
- Probability of a false positive $\approx 0.007 \approx 0.7\%$
  - 1 out of 142
Hash coding with allowable errors

- On the one hand:
  - Save space
  - Very fast query

- On the other hand:
  - Not deterministic
  - May yield false positives (though never false negatives)

- Trade-off: errors are allowable → hash area can be made small
Another use-case: IP Traceback

- Not only good packets travel through the Internet
  - Malicious packet: trace back its route

- Naive idea: each router stores the packets it transmits for some period of time
  - Victimized computer can query routers above it
    - Space-consuming
    - Storing packets: target for attack

- Instead: store its digest using a Bloom filter
  - Trade certainty for efficiency and space
  - Have you seen x? YES/NO
Outline

1. Multicast in publish/subscribe networks
   1. Pub/sub network architecture
2. Bloom filter basics
   1. What is a Bloom filter?
   2. False positive probability
3. Forwarding on Bloomed link identifiers
   1. Bloom-filter based multicast forwarding method
   2. Limitations
4. Concluding remarks
Basic Forwarding Method

- No end-to-end addresses
- Identify links (instead of nodes)

- The topology system constructs forwarding identifiers
- Constructs a multicast forwarding tree
- Each node makes a forwarding decision
Multicast forwarding using Bloom filters

1. Assign LinkIDs
   - Two identifiers = **LinkIDs** for each link:
     - Between nodes A and B: \(\overrightarrow{AB}\) and \(\overrightarrow{BA}\)
   - Each LinkID can be locally assigned
     - Low probability of duplicates
   - LinkID: \(m\)-bit long name with \(k\) bits set to 1
     - Typically \(k << m\)
     - With appropriate \(k\) and \(m\) the LinkIDs are statistically unique
     - E.g. \(m=248, k=5\)
     - No. of LinkIDs = \(m!/(m-k)! \approx 9*10^{11}\)
Forwarding tree

2. Create a multicast tree

- **Topology system:** graph of the network
  - LinkIDs and connectivity
- **Request:** determine a forwarding tree
  - Heuristic based on shortest paths
  - Spanning tree
- **Source-specific**
  - Even for the same set of subscribers
  - Different sources yield different forwarding trees
Encoding & Forwarding

3. Encoding
   - Forwarding tree OK
   - Add its links to a Bloom filter
   - Place it in the packet header = *in-packet Bloom filter*

4. Forwarding at a node

   **Input:** LinkIDs of outgoing links, in-packet Bloom filter in packet header

   **Foreach** LinkID of outgoing interface **do**
   
   if in-packet Bloom filter AND LinkID == LinkID then
     Forward packet on the link;
   
   end

end
Multicast Example
Feasibility of the approach

- Forwarding efficiency
  \[ f_{we} = \frac{\#\text{Links on shortest path tree}}{\#\text{Links during delivery}} \]

- One in-packet Bloom filter can address up to 23 subscribers
  - \( \approx 32 \) links
  - \( f_{we} > 90\% \)

- Reasonable performance up to 20 subscribers

- Why not more?
  - Overfilled Bloom filters
Supporting Larger Trees

1. Send multiple packets
   - Several smaller multicast trees instead of one large
   - Keeps the in-packet Bloom filters’ fill factor reasonable

   - Several delivery trees instead of one
   - Delivery trees will overlap
   - Fine-tuning: less bandwidth waste than for one large tree
Supporting Larger Trees

2. Multi-Stage Bloom filters

- Instead of one large filter: use a series of stage filters
- **Stage filter**: contains forwarding information about the links at a distance of $h$ hops from the source
  - Offer information about the topology in the header
  - Should be deleted one by one
- A forwarding tree of $h$ links is represented by $h$ stage filters
  - $i^{th}$ filter contains links that are at a distance of $i$ hops from the source
Supporting Larger Trees

- Gradually delete the unnecessary stage-filters at each stage
  - Less and less overhead along the way
- Optimize the filter length at each stage
  - Results in results in varying sized stage filters.
  - For identifying filter boundaries: store the length of each filter in the header
  - To indicate boundaries for an $m$-bit long filter:
    1. Write $\lceil \log_2(m) \rceil - 1$ zero bits;
    2. Followed by the binary representation of $m$
Multi-Stage Bloom Filter Example

- Traditional Bloom filter with false positives

in-packet Bloom-filter: 110111
Multi-Stage Bloom Filter Example

- Multi-stage false positive free Bloom filter

![Diagram of a multi-stage Bloom filter with stages and filters]

Stage 1 filter: \[110011\]

Stage 2 filter: \[110111\]
References

