Lecture 4: Routing design - forwarding & lookup

- S/W stack

Last lecture: overview of router design - trade-offs in speed vs. design

- components
- backplane

Today: More on router's data plane.

Things that happen before switching across the backplane.

- Lookup and forwarding classification.

Contact: addressing

CIDR: previously - classes.

- lead to inefficient use of address space
- multiple class C's -> routing table explosion
- CIDR allows aggregation
- no rigid class number

Implications: longest prefix matching.

How to do LPM really fast:

1. Trie: worst case \(O(n)\) lookups
2. CAMs: large, power-consuming expensive, not very dense
3. Protocol-based approaches: tags, VCs
   - Ingress router needs to do full lookup
   - Also, need global tag agreement and secure protocols
4. Caching: not very effective for backbones when tree is insufficient locality
5. Binary search over entries: \(\log(n)\) but efficient storage
Requirements: (1) few memory accesses per packet
(2) small amount of storage.

Ideas: A hash table per prefix length.

Linear search $\rightarrow \log_2 W$ (start for largest prefix length)

Binary search:

<table>
<thead>
<tr>
<th>Length</th>
<th>Hash Tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
</tbody>
</table>

$O(W \log_2 W)$

Binary search $\rightarrow$ non-trivial to get it to work as markers are needed.

<table>
<thead>
<tr>
<th>Length</th>
<th>Tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Add marker! 0</td>
</tr>
<tr>
<td>2</td>
<td>Search for 111</td>
</tr>
<tr>
<td>3</td>
<td>Fail</td>
</tr>
</tbody>
</table>

Marker Match $\rightarrow$ check lower half.
No Match $\rightarrow$ check upper half.

Note: How many markers: only in prefix lengths that are likely to be traversed in binary search

$\sim O(\log W)$
Marker problems → backtracking.

1

2 (1) 00

3 111

Suppose 110 arrives → should match (1) but search provokes to level 3 and fails. Need to backtrack to level 2 and try out top half. → worst-case backtracking complexity → O(w).

How to solve backtracking.

For each marker, remember the longest matching prefix → value of LMP of marker M.

(1) → (1). 1

Before going down the path suggested by a marker → remember M.LMP.

No need to backtrack as the result of backtracking is stored in M.LMP.

How to compute M.LMP → precomputation when lookup table is constructed.

Find algorithm: (W log W) space and log (W) speed.

Also on next page.

Some optimizations:

1. Precompute search paths to maximize likelihood of finding a match.

2. Cross-hint caching hints pick prefix length with most entries as the root of the binary search → shown to work well in practice.
Final algorithm:

Binarysearch(D) /* search for address D */
Initialize search range R over the whole array of lengths L
Initialize LMP found so far to NULL.

While R is not empty
    Let i correspond to the middle level in R
    Extract the first \( \log \text{length}(L(i)) \) bits of D into D'
    M := Search(D', L(i).hash)
    if M is null then R := upper half of R
    else if M is a prefix and not marker
        then LMP := M.LMP; break
    else if M is just a marker or marker + prefix /
        LMP := M.LMP
    R := lower half of R

Endwhile

In practice \( \log \approx 20 \therefore \log \text{length}(D) \approx 4 \approx 4.5 \)

Assum. tree optimizations bring explicated case lookup time to 2.

Can we do it in one clock cycle? Hard:

Issues: 1. memory lookup per round \( ightarrow \) lookups are dependent
        and have to be done serially.

2. Idea: bloom filters.

\[ \text{prefix lengths} \quad \text{BF1} \quad \text{summarizes} \quad L1 \]
\[ \text{BF2} \]
\[ \text{BF3} \]
Summary:

- For hash functions: \( h_1 \ldots h_k \)
- For entry \( e \):
  \[ h_1(e) \ldots h_k(e) \] set those bits in BF to 1
  (some may already be set to 1 by others)

Look up \( e \):
- Compute \( h_1(e) \ldots h_k(e) \);
- Return yes if all are 1.

- Likelihood for false positives: denominator is number of
  \( n \) \( \hat{n} \)
  \( \hat{n} \) \( n \) \( \approx m/k \)
  \( m \) \( n \) \( \approx m \), \( m \) \( n \)

\( k \rightarrow \) too small is too bad
\( k \rightarrow \) too big is bad as well as many bits get taken away.

It is possible to tune a BF to achieve a target FP rate.

- Points on course Web page for

\[
f = \left( 1 - \left(1 - \frac{1}{m} \right)^{mk} \right)^{\binom{k}{2}}
\]

Optimal: \( k = \frac{m}{n} \ln 2 \)

\( \text{fp} = \frac{1}{2}^k \).

Back to the algo:
- Check all BF's — no time consumed here
- Start with longest prefix length
- And see if hash table has entry
  - FPs impact if entry exists or not
- Use BF's — if bad -> use older idea