Lookup Problem

- Identify the output interface to forward an incoming packet based on its destination address
- Routing (forwarding) tables summarize information by maintaining prefixes
- Route lookup → find the longest prefix in the table that matches the packet destination address
Example

- Packet with destination address 12.82.100.101 is sent to interface 2, as 12.82.100.xxx is the longest prefix matching packet’s destination address.
Patricia Tries

- Use binary tree paths to encode prefixes

- Advantage: simple to implement
- Disadvantage: one lookup may take $O(m)$, where $m$ is number of bits (32 in the case of IPv4)
Lulea’s Routing Lookup Algorithm

- Minimize number of memory accesses
- Minimize size of data structure
  - Small size allow to fit entire data structure in the cache (why do you care about size?)
- Solution: use a three level data structure
First Level: Bit-Vector

- Cover all prefixes down to depth 16
- Use one bit to encode each prefix
  - Memory requirements: $2^{16} = 64$ Kb = 8 KB
First Level: Pointers

- Maintain 16-bit pointers to (1) next-hop (routing) table or (2) to two level chunks
  - 2 bits encode pointer type
  - 14 bits represent an index into routing table or into an array containing level two chunks
- Pointers are stored at consecutive memory addresses
- Problem: find the pointer
Example

Routing table

Level two chunks

Problem: find pointer

bit vector 1 0 0 0 1 0 1 1 1 0 0 0 1 1 1 1 ...

pointer array

0006abcd

000acdef
Code Word and Base Indexes Array

- Split the bit-vector in **bit-masks** (16 bits each)
- Find corresponding bin-mask
- **How?**
  - Maintain a 16-bit **code word** for each bit-mask (10-bit value; 6-bit offset)
  - Maintain a **base index** array (one 16-bit entry for each 4 code words)

![Diagram of code word and base index array]

number of previous ones in the bit-vector
First Level: Finding Pointer Group

- Use first 12 bits to index into code word array
- Use first 10 bits to index into base index array
First Level: Encoding Bit-masks

- Observation: not all 16-bit values are possible
  - Example: bit-mask 1001… is not possible (why?)
- Let a(n) be number of bit-masks of length $2^n$
- Compute a(n) using recurrence:
  - $a(0) = 1$
  - $a(n) = 1 + a(n-1)^2$
- For length 16, we get only 677 possible values for bit-masks
- This can be encoded in 10 bits
  - Values $r_i$ in code words
- Store all possible bit-masks in a table, called maptable
First Level: Finding Pointer Index

- Each entry in Maptable is an offset of 4 bits:
  - Offset of pointer in the group
- Number of memory accesses: 3 (7 bytes accessed)
First Level: Memory Requirements

- Code word array: one code word per bit-mask
  - 64 Kb
- Based index array: one base index per four bit-mask
  - 16 Kb
- Maptable: 677x16 entries, 4 bits each
  - ~ 43.3 Kb
- Total: 123.3 Kb = 15.4 KB
First Level: Optimizations

- Reduce number of entries in Mapttable by two:
  - Don’t store bit-masks 0 and 1; instead encode pointers directly into code word
  - If r value in code word larger than 676 → direct encoding
  - For direct encoding use r value + 6-bit offset
Levels 2 and 3

- Levels 2 and 3 consists of chunks
- A chunk covers a sub-tree of height 8 \(\rightarrow\) at most 256 heads
- Three types of chunks
  - Sparse: 1-8 heads
    - 8-bit indices, eight pointers (24 B)
  - Dense: 9-64 heads
    - Like level 1, but only one base index (< 162 B)
  - Very dense: 65-256 heads
    - Like level 1 (< 552 B)
- Only 7 bytes are accessed to search each of levels 2 and 3
Limitations

- Only $2^{14}$ chunks of each kind
  - Can accommodate a growth factor of 16
- Only 16-bit base indices
  - Can accommodate a growth factor of 3-5
- Number of next hops $\leq 2^{14}$
This data structure trades the table construction time for lookup time (build time < 100 ms)
- Good trade-off because routes are not supposed to change often

Lookup performance:
- Worst-case: 101 cycles
  • A 200 MHz Pentium Pro can do at least 2 millions lookups per second
- On average: ~ 50 cycles

Open question: how effective is this data structure in the case of IPv6?
Classification Problem

- Classify an IP packet based on a number of fields in the packet header, e.g.,
  - source/destination IP address (32 bits)
  - source/destination port number (16 bits)
  - TOS byte (8 bits)
  - Type of protocol (8 bits)

- In general fields are specified by range
Example of Classification Rules

- Access-control in firewalls
  - Deny all e-mail traffic from ISP-X to Y

- Policy-based routing
  - Route IP telephony traffic from X to Y via ATM

- Differentiate quality of service
  - Ensure that no more than 50 Mbps are injected from ISP-X
Characteristics of Real Classifiers

- Results are collected over 793 packet classifiers from 101 ISPs, with a total of 41,505 rules
  - Classifiers do not contain many rules: mean = 50 rules, max = 1734 rules, only 0.7% contain over 1000 rules
  - Many fields are specified by range, e.g., greater than 1023, or 20-24
  - 14% of classifiers had a rule with a non-contiguous mask
  - Rules in the same classifier tend to share the same fields
  - 8% of the rules are redundant, i.e., they can be eliminated without changing classifier’s behavior
Example

- Two-dimension space (i.e., classification based on two fields)
- Complexity depends of the layout (i.e., how many distinct regions are created)
Hard Problem

- Even if regions don’t overlap, with $n$ rules and $F$ fields we have the following lower-bounds
  - $O(\log n)$ time and $O(n^F)$ space
  - $O(\log ^{F-1} n)$ time and $O(n)$ space
Simplifying Assumptions

- In practice, you get the average not the worst-case, e.g., number of overlapping regions for the largest classifier 4316 vs. theoretical worst case $10^{13}$
- The number of rules is reasonable small, i.e., at most several thousands
- The rules do not change often
Recursive Flow Classification (RFC) Algorithm

- Problem formulation:
  - Map $S$ bits (i.e., the bits of all the $F$ fields) to $T$ bits (i.e., the class identifier)

- Main idea:
  - Create a $2^S$ table with pre-computed values; each entry would contain the class identifier
    - Only one memory access needed
  - …but this is impractical → require huge memory
RFC Algorithm

- Use recursion: trade speed (number of memory accesses) for memory footprint
The RFC Algorithm

- Split the F fields in chunks
- Use the value of each chunk to index into a table
  - Indexing is done in parallel
- Combine results from previous phase, and repeat
- In the final phase we obtain only one value
Example of Packet Flow in RFC
Complete Example

- Four fields → six chunks
  - Source and destination IP addresses → two chunks each
  - Protocol number → one chunk
  - Destination port number → one chunk

Table 6:

<table>
<thead>
<tr>
<th>Rule#</th>
<th>Chunk#0 (Src L3 bits 31..16)</th>
<th>Chunk#1 (Src L3 bits 15..0)</th>
<th>Chunk#2 (Dst L3 bits 31..16)</th>
<th>Chunk#3 (Dst L3 bits 15..0)</th>
<th>Chunk#4 (L4 protocol) [8 bits]</th>
<th>Chunk#5 (Dst L4) [16 bits]</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>0.83/0.0</td>
<td>0.77/0.0</td>
<td>0.0/0.0</td>
<td>4.6/0.0</td>
<td>udp (17)</td>
<td>*</td>
<td>permit</td>
</tr>
<tr>
<td>(1)</td>
<td>0.83/0.0</td>
<td>1.0/0.255</td>
<td>0.0/0.0</td>
<td>4.6/0.0</td>
<td>udp</td>
<td>range 20 30</td>
<td>permit</td>
</tr>
<tr>
<td>(2)</td>
<td>0.83/0.0</td>
<td>0.77/0.0</td>
<td>0.0/255.255</td>
<td>0.0/255.255</td>
<td>*</td>
<td>21</td>
<td>permit</td>
</tr>
<tr>
<td>(3)</td>
<td>0.0/255.255</td>
<td>0.0/255.255</td>
<td>0.0/255.255</td>
<td>0.0/255.255</td>
<td>*</td>
<td>21</td>
<td>deny</td>
</tr>
<tr>
<td>(4)</td>
<td>0.0/255.255</td>
<td>0.0/255.255</td>
<td>0.0/255.255</td>
<td>0.0/255.255</td>
<td>*</td>
<td>*</td>
<td>permit</td>
</tr>
</tbody>
</table>
\[ \text{indx} = c_{10} \times 5 + c_{11} \]

\[ \text{indx} = c_{02} \times 6 + c_{03} \times 3 + c_{05} \]
\[ \text{indx} = c10 \times 5 + c11 \]
RFC Lookup Performance

- Dataset: classifiers used in practice
- Hardware: 31.25 millions pps using three stage pipeline, and 4-bank 64 Mb SRAMs at 125 MHz
- Software: > 1million pps on a 333 MHz Pentium
RFC Scalling

- RFC does not handle well large (general) classifiers
  - As the number of rules increases, the memory requirements increase dramatically, e.g., for 1500 rules you may need over 4.5 MB with a three stage classifier

- Proposed solution: adjacency groups
  - Idea: group rules that generate the same actions and use same fields
  - Problems: can’t tell which rule was matched
Summary

- Routing lookup and packet classification → two of the most important challenges in designing high speed routers
- Very efficient algorithms for routing lookup → possible to do lookup at the line speed
- Packet classification still an area of active research
- Key difficulties in designing packet classification:
  - Requires multi-field classification which is an inherently hard problem
  - If we want per flow QoS insertion/deletion need also to be fast
    - Harder to make update-lookup tradeoffs like Lulea’s algorithm
RFC Algorithm: Example

- **Phase 0:**
  - Possible values for destination port number: 80, 20-21, >1023, *
    - Use two bits to encode
    - Reduction: 16→2
  - Possible values for protocol: udp, tcp, *
    - Use two bits to encode
    - Reduction: 8→2

- **Phase 1:**
  - Concatenate from phase 1, five possible values: {80,udp}, {20-21,udp}, {80,tcp}, {>1023,tcp}, everything else
    - Use three bits to encode
    - Reduction 4→3