Packet classification

Outline

The classification problem
Types of address (part 1: Classful)
Classification variations
Exact match classification
Lookup tables
Linked lists
Hashing
CAMs
Types of address (part 2: Classless)
Partial match classification
Exact match techniques revisited
Ternary CAMs
Tries, PARTICIA trees

Student classification: problem

How can a lecturer rapidly look up student data of $E$ bits? e.g. to record attendance of students at a lecture ($E=1$)

Each student is identified by a 7 digit student number ($K=7$)
- 10^7 possible numbers
- only 78 students in class ($D=78$)
- numbers of students in class are correlated, since they are likely to have enrolled at similar times and numbers are allocated sequentially over time (e.g. many beginning 310...)

Student classification: solutions

A1: Use student number to index a table in RAM.
Wastes RAM since 78 << 10^7

A2: Convert student number into smaller number to index a table. e.g. choose least significant 3 digits.
More efficient with RAM but may have collisions (Hashing)

A3: Special associative memory that can simultaneously match all stored words with supplied value. (CAMs)

A4: List them in order (overhead when class changes) &:
- search sequentially
- binary search

A5: Other data structures (linked lists, tries, etc)
Hashing

Definition: Mathematical mapping from large value to smaller one (ideally $K \rightarrow \lceil \log_2 D \rceil$). Smaller value can then be used to directly index a table. If table contains an entry, check if it matches full key.

Issues: Mapping function should minimise collisions ("hash bash") in which multiple large values being used map to the same smaller value.

To add an entry to the table:
1. Check whether the hash of the new entry collides with an existing value.
2a. If not, add new entry directly to table.
2b. If there is a clash, update the existing entry s.t. it points to classifier used to resolve collisions (e.g. linked list or CAM). Add new entry to that classifier.

e.g. Hashing of student numbers

Using 3 least significant digits
1000 entry table:
- 74 entries for unique IDs
- 2 for collisions

Address IDs

134 3067134
188 3108188
203 2227203, 314203
204 -
205 -
206 3050206
207 -
208 -
209 3081209
217 3120, 3125217
231 3116

Least significant 2 digits of sum of pairs of digits
100 entry table:
- 36 unique entries (about half)
- 18 for collisions (worst: 4)

Address IDs

4 3 11 03 87
6 3 08 09 08, 3 11 97 95
7 3 08 05 91, 3 11 62 31
8 -
9 3 08 02 96, 3 10 95 01
10 3 06 73 28, 3 12 17 78
11 3 03 93 12, 3 05 94 11, 3 07 00 01

Compare 7.4% or 36% utilisation with 0.78E-3% for single lookup

Example hash functions

- Subset of the key bits
  e.g. for student numbers or MAC addresses:
  - Least significant bits: good (differentiate individuals)
  - Most significant bits: bad (students: similar for same year; MAC: LAN may have many devices from one manufacturer)

Usually apply a more sophisticated function s.t. hashing process is insensitive to format and it is difficult to find a set of addresses that cause collisions

- Arithmetic operations
  - e.g. add digits in student number
  - more complex operations, e.g. CRC

CRCs as hashing functions

CRCs make decent hashing functions for Ethernet switches:

- Scramble output well
- Need to calculate anyhow. Calculate CRC over whole frame, but take intermediate result of calculation (after processing first 6 or 12B of header) for hash function.
- Important keys (DA/SA) are at start of frame, so CRC won’t vary because of preceding fields (unlike 802.11 – hash depends on duration)

Ethernet frame format:

<table>
<thead>
<tr>
<th>Destination Address</th>
<th>Source Address</th>
<th>Type</th>
<th>Len</th>
<th>Data</th>
<th>CRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-4-8-5-6-2-1-1-1-1-1</td>
<td>4-4-8-5-6-2-1-1-1-1-1</td>
<td>5LP</td>
<td>2B</td>
<td>46-1500B</td>
<td>32B</td>
</tr>
</tbody>
</table>

- E 802.11 frame header:
  - 4 destination address (DA) bytes
  - 4 source address (SA) bytes
  - 2 type (TP) bytes
  - 11 length (LEN) bytes
Content Addressable Memories

Each word of memory contains:
• $K$ bits of label storage and comparison logic
• and possibly data

Matching process:
1. Key is distributed to all words simultaneously.
2. Comparison operations are done in parallel.
3. Words with matching labels generate signal.

Figure shows a CAM with
• 4 words with 3b labels and 1b data
• $D \leq 4$, $K=3$, $E=1$

CAM processing of match signals

Processing of match signals:
• OR them to determine if there is any match (e.g. packet filtering), CAM may output data associated with word.
• Encode to index separate table of data
• Arbitrate (e.g. lowest word) multiple matches (e.g. when searching for empty word, or partial-match searching)

Example CAM: MUAA8K80

20ns clock speed;
• 32b I/O ⇒ multiple cycles
• 80-bit width with programmable CAM/RAM partition
For bridges: Auto-learn Internal aging with 9-bit time stamp
• Synchronous port for high speed packet processing
• Asynchronous port for table maintenance

Package: 160 PQFP
Recall also, the RTL 8308 chip, used in the D-link DES-1008D switch, has a 128-entry CAM to accommodate hash bashing.

Are CAMs the ideal classifier?

Sample specs:

<table>
<thead>
<tr>
<th>Device</th>
<th>Capacity</th>
<th>Clock Speed (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Music Semiconductors</td>
<td>SK x 106</td>
<td>50</td>
</tr>
<tr>
<td>MUAA8K80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDT 75K8100</td>
<td>128K x 72b</td>
<td>133</td>
</tr>
</tbody>
</table>

64B Ethernet frames @ 100ns ⇒ 2M frames/s ⇒ clock speed of 50MHz sounds fine, except multiple clock cycles required to feed data in/out

• Comparison logic for each cell
• Relatively low density (bits/unit area)
• e.g. 46 transistors for SRAM cell, 11-15 for CAM ⇒ more space/bit for CAMs
• High power consumption
  e.g. 2W for 8Mb SRAM @ 200MHz; 7W for 2Mb CAM @ 50MHz
• Speed limited by arbiter
• Niche memory ⇒ not improved through competitive pressures (ala DRAM)
Hybrid classifiers

Try using one classifier & if that fails, revert to another, e.g.

- Hash then CAM (e.g. as in RTL8308 chip used in D-link DES-1008D):
  1. Try hashing first.
  2. Resolve hash bashes by storing colliding labels in a CAM.
     Small CAM resolves rare hash bashes.
- CAM then trie:
  1. Try the CAM first (fast)
  2. If CAM is inconclusive, then use a trie (described shortly) that
     stores all labels.
     Common cases are processed fast. Trie (large but slow) deals with
     exceptional cases.

Problems with classfull addresses

- Free addresses become scarce as Internet becomes popular
  \( \Rightarrow 128b \) for IPv6
- Poor utilisation: Organisations \( O \) with \( 2^{9}+1 \) hosts on one
  network needs Class B space and uses only \( 1/256 \)th
  - Could use 2 separate Class C networks, but impacts
    routers...
  - Router tables becoming too large (2M Class C nets)

Solution: Classless InterDomain Routing (CIDR)

Classless Inter Domain Routing

CIDR allows network prefixes of arbitrary length

Different way of indicating which bits identify network, and
which bits identify host.

- No longer indicated by address class
- Instead, indicated by a separate network mask (=1 in
  network ID bits)
- Routers exchange masks with other reachability info.

\[
\begin{align*}
\text{Address:} & \quad 10010101.10101011.00100100.001100 \quad (149.171.36.48) \\
\text{Mask:} & \quad 11111111 11111111 11111110 00000000
\end{align*}
\]

could have been class B

used with 96 for “host”
CIDR prefix lengths

Usually the mask covers contiguous bits, starting with the first bit, so it can be defined by a prefix length.

\[ 149.171.36.48/23 = 147.171.36 + 0.48 \text{(host)} \]
\[ 149.171.37.48/23 = 147.171.36 + 1.48 \text{(host)} \]

Prefix length indicates significance of trailing 0s, e.g.

\[ 192.16.32/2 = 192.16.0010 \text{hh (h=bit identifying host)} \]
\[ 192.16.32/22=192.16.001000 \text{hh} \]

Routing protocols exchange network addresses + prefix lengths.

Address aggregation: Scenario

Router has moderate number of ports (e.g. 4-64) & different networks may be reachable through same port.

E.g. two organisations

- \( O \) with network 192.16.32/23 through port \( P \)
- \( Q \) with network 192.16.34/23 through port \( P \)

Forward aggregate 192.16.32/22 through port \( P \)

Address aggregation: Implementation

Aggregation reduces the number of router table entries.

Ideally assign topologically-adjacent organisations numerically-adjacent addresses; e.g. APNIC.

Organisations may move (e.g. \( Q \) shifts headquarters overseas or changes network service provider) ⇒ aggregation exceptions.

E.g. 192.16.32/19 through port \( P \), except 192.16.34/23 through port \( R \)

Search for longest prefix match (aka "best prefix match"): longer prefix is more specific and so prefer to route there (give it priority).

Prioritisation of rules

Classification rules may intersect ⇒ need to prioritize, e.g.: Router: Longest-prefix matching

Firewall: Filtering traffic may have priority over send certain traffic on matching path, e.g.:

1. Block traffic from network \( N \)
2. Forward traffic to network \( M \) through port \( P \)

Packet from \( N \) to \( M \) should be blocked.
Partial matching

Rules may be based on **exact match or partial match** against key.

Partial match may be based on:
- **prefix** – common, e.g. IP network classification
- **selected fields** – less common, e.g. gateway/firewall may have multiple rules, e.g.:
  - some depend on **SA** (e.g. block all from spammer)
  - others don’t care about **SA** (e.g. block all requests to insecure finger service port 79)
- some depend on Class Of Service (e.g. telnet traffic on port 23 to low delay queue, FTP traffic on port 20 to high-throughput queue), others don’t care.

Can’t co-locate don’t care fields as a suffix ⇒ can’t use longest-prefix matching techniques.

Examples of classification on selected fields

- Some depend on **SA** (e.g. block all from spammer), others don’t care about **SA** (e.g. block all requests to insecure finger service port 79)
- Some depend on Class Of Service (e.g. telnet traffic on port 23 to low delay queue, FTP traffic on port 20 to high-throughput queue), others don’t care.

When matching on selected fields, can’t co-locate don’t care fields as a suffix ⇒ can’t use longest-prefix matching techniques.

Partial matching with the preceding classifiers

- **Tables**
  - Hard to update – e.g. to install rule with mask that doesn’t care about $b$ bits may have to update $2^b$ table entries.
  - No benefit from aggregation
- **Lists**
  - OK if place longest prefix first, can’t sort to expedite search
  - Still doesn’t scale well to many entries.
- **Hashing**
  - Only suitable for exact matches – not longest prefix
- **CAMs** – add ternary functionality …
Ternary CAMs

Ternary comparison: 0, 1, X ("Don’t care" = 0 or 1)

Don’t care bits generally specified as a mask (e.g. 1="Do care", 0="Don’t")

Key mask
apply to all words; iterative search
- e.g. mask = 24 ones, then 23 ones, then ...

Word mask
word-specific, stored with label

CAMs are good when excluded bits needn’t form a suffix, e.g. gateway

Tries

Trie (from retrieve):
- "a tree where each node corresponds to a string that is defined by the path to that node from the root."

Rule Prefix
- a 0*
- b 01000*...
- i 11110*
- j 1111*
- k 12345

Applying to partial match: Locate “don’t care” at end of key.
- OK for CIDR and address aggregation, but not for gateway filtering

Implementing tries in RAM

Each node stored in one word, containing at most 3 ptrs:
- next rule if bit is 0
- next rule if bit is 1
- rule if this is last matched bit. (Only need last pointer if other 2 aren’t leaves, e.g. not needed for nodes 8 and 9.)

Start search at node 1, continue until either:
- match leaf node, e.g. 011 matches c
- mismatch leaf node, then use previous best match, e.g. 01001 matches a en route to mismatching b.

Optimising tries

Hopefully fit recently-used parts of trie in processor’s cache
- Lengthening the "stride" with multi-bit tries →
- Collapsing nonbranching branches →
Implementing tries

Implement nodes as tables that point to lower branches
• "stride" size = number of bits used to branch out from each node, e.g. stride of 1 = binary
• Larger stride reduces number of nodes traversed, but requires larger tables that may be sparsely occupied (with memories increasing in size faster than increasing in speed, wasting some space for number of accesses makes sense)

PATRICIA\(^\dagger\) trees

Related to tries
Observation: When keys are sparse, many trie nodes have only one descendent
Solution: Rather than use every bit of key to determine subtree, record in node which bits should be used.
Provides "path compression" by recording only genuine branches.
Standard implementation under BSD Unix (so claims Ruiz-Sanchez – bonus mark if you can identify the Linux algorithm)

Prefixes

- a. 0*
- b. 01000*
- c. 011*
- d. 1*

To start search, look at bit 1
To search further, look at bit 3; skip bit 2.

Resources

Relevant sections of Keshav:
• Chapter 10: Addressing
• pp. 176-8: classification ("port mappers")