How to live with IP forever

(or at least for quite some time)

IPv6 to the rescue!

• Solves all problems with IPv4

• Standardized during the 1990’s
  – Final RFC in 1999
IPv4 vs IPv6

- 32-bit addresses
- IPSec optional
- No flow identifier
- Fragmentation in routers
- Header checksum
- IP options
- Might be 576 bytes
- Uni-, Broad-, Multicast
- NAT a requirement
- ARP/RARP

- 128-bit addresses
- IPSec mandatory
- Flow identifier for QoS
- Fragmentation in end hosts
- No header checksum
- Extension headers
- Might be 1280 bytes
- Uni-, Multi, Anycast
- No need for NAT
- Neighbour discovery, ICMPv6

The IPv4 Header

```
+----------------+----------------+-----------------+-----------------+
<table>
<thead>
<tr>
<th>Ver</th>
<th>IHL</th>
<th>Service Type</th>
<th>Total Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>Flags</td>
<td>Fragment Offset</td>
<td>Time to Live</td>
</tr>
<tr>
<td>Protocol</td>
<td>Header Checksum</td>
<td></td>
<td>Options and Pad</td>
</tr>
<tr>
<td>32 bit Source</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 bit Destina-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

0 bits  4  8  16  24  31

- Changed
- Removed
The IPv6 Header

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>12</th>
<th>16</th>
<th>24</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>Class</td>
<td>Flow Label</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Payload Length</td>
<td>Next Header</td>
<td>Hop Limit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128 bit Source Address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128 bit Destination Address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IPv6 addresses

- 64 bits for location
  - Specifies the location in the network
  - Can change if the node is moving
  - Used for forwarding
- 64 bits for identification
  - Specifies the interface on the Internet
  - Does not change when node is moving
  - Can be IPv4 address, EUI-64, EUI-48, ...
  - Used for process multiplexing
- Multicast addresses an exception!
- No need for netmask or subnet identifier
  - Simplifies automatic configuration without DHCP
- Cryptographically protected addresses (RFC3972)
Transition mechanisms

- Dual-stack solution
- Tunneling
  - IPv6 packets carried as data in IPv4 packets
- Translation methods
  - Network layer: Rewrite IP datagrams
  - Transport layer: Rewrite TCP segments
  - Application layer: Application gateways

So, IPv6 to the rescue - or?

- Addresses shortcomings of IPv4
  - Is that enough?
- Node heterogeneity in the Internet increases
  - Slowest nodes and networks get slower
  - Fastest nodes and networks get faster
- Complexity and overhead still an issue
- Care must be taken to address concerns
Algorithmic engineering to the rescue!

- Concerns about IP in future devices
- Avoid Dual stacks and dual infrastructures
- Key application for cellular: VoIP
  - Real-time constraints
  - Short packets with sound clips
  - Interactive voice
  - Low-power devices where energy is a concern

Algorithmic engineering

1. Define the problem
2. Construct a theoretical model
3. Solve the problem in the theoretical model
4. Implement the solution
5. Evaluate the performance and try to improve it using information lost in the theoretical model
6. If the theoretical model did not capture the behavior of the reality, refine the model and continue from 3)
Example

• The problem definition:
  
  Write a C function to search a sorted array of 16 unsigned integers for a given query value.

Theoretical model

• Let \( A = \{a_0, a_1, ..., a_{15}\} \)

• For any \( 0 < i \leq 15, a_{i-1} < a_i \)

• Given a query element \( x \), is there an \( i, 0 \leq i \leq 15 \), such that \( x = a_i \)?
Theoretical solution

- Strawman approach: Linear search
  - On average 8 comparisons

- Better idea: Binary search
  - Use the fact that $A$ is sorted
  - Compare to median to discard half the elements
    - 16 candidates -> 8 candidates after first comparison
    - 8 candidates -> 4 candidates after second comparison
    - 4 candidates -> 2 candidates after third comparison
    - Last candidates both compared
    - A total of 5 comparisons

- Linear search about 70% slower than linear search

Implementation time!

```c
inline uint LSearch16(unit *a, uint x) {
    uint i;
    for (i=0; i < 16; i++) {
        if (x == a[i]) return 1;
    }
    return 0;
}

inline uint Bsearch16(uint *a, uint x) {
    uint lb = 0; ie = 16, im;
    while (lb != ie) {
        im = (lb + ie) / 2;
        if (x <= a[im]) {
            ie = im;
        } else {
            lb = im + 1;
        }
    }
    return x == a[im] || x == a[im+1];
}
```
Evaluation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average CPU cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSearch16</td>
<td>44.9</td>
</tr>
<tr>
<td>BSearch16</td>
<td>57.7</td>
</tr>
</tbody>
</table>

Why is binary search slower?!

- **Comparisons**
  - Two in `while` loop
  - Two in `for` loop
  - BSearch16 should be slightly faster

- **Arithmetics**
  - Division and addition in BSearch16
  - More complex `IF` statement in BSearch16
Refinement: Remove loop

```c
inline uint QSearch16(uint *a, uint x) {
    a += (x >= a[8]) << 3;
    a += (x >= a[4]) << 2;
    a += (x >= a[2]) << 1;
    return x == a[0] || x == a[1]
}
```

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average CPU cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSearch16</td>
<td>44.9</td>
</tr>
<tr>
<td>BSearch16</td>
<td>57.7</td>
</tr>
<tr>
<td>QSearch16</td>
<td><strong>18.8</strong></td>
</tr>
</tbody>
</table>

Algorithmic engineering is about...

- Taking hardware into account
- Bending (but not breaking) the rules
  - Rules? Yes, there are rules!
- Thinking outside the box
  - There is no box - it’s just imaginary.
One upon a time, it was told:

- IP over wireless will never work
  - Too much overhead
  - Too heavy processing
  - Too costly to use IPv6
- IPv6 will never take off
  - Too costly for home consumers
  - IP address lookup bottleneck is CPU, not memory

I will now (briefly) show how to:

- Compress TCP/IP headers down to one (1) bit
- Do packet lookups really really fast
- Common factor:
  - Algorithmic engineering
Header compression – basic idea

- Reduce headers over one link
  - No need for layering and IP routing support
- TCP/IP headers reconstructed after this link
- Establish a connection between to HC peers
  - Peers share state representing different sessions
  - Compressed header includes state identifier
  - To decompress, state is expanded to full TCP/IP headers
- Remove all information that is static, redundant or possible to deduce/calculate

---

Overhead example: RTP packets

<table>
<thead>
<tr>
<th>IPv4</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vers</td>
<td>Hlen</td>
<td>TOS</td>
<td>TOS</td>
<td>TOS</td>
<td>TOS</td>
</tr>
<tr>
<td>Identification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TTL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Protocol</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Destination address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source port</td>
<td></td>
<td>Dest. port</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Length</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timestamp</td>
<td></td>
<td>Sequence no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSRC Identifier</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSRC Identifier</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4 bytes

Legend

- STATIC
- INFERRED
- CHANGING-RARELY
- CHANGING-OFTEN

Appr. 30 of 40 octets are rather static!
Compressing / Decompressing RTP

- Full reconstruction of RTP/UDP/IP headers
- Overhead of 4-6 bytes per packet
- Can we do better?!

Breaking the rules

- Remove checksum entirely
  - Replace with shorter CRC
- Remove identification entirely
  - Replaced by state identifier
- Reduce the timestamp
  - Only represent delta intervals
- Reduce the sequence number
  - Use less bits, use delta coding
- Consider not only changes between full headers but also between compressed ones!
- ...ends up at 1-2 bytes per packet
  - Can we do better?!
1-bit Header compression

• Rely on underlying link layer
  – Strong checksum => No CRC needed
  – Ordering => No Sequence# needed
  – Constant sampling rate => No timestamp needed
• We do not need ANY TCP/IP headers!!!
  – Well, except for when something differs
• One bit to indicate if something differs
  – Slap on another 7 bits to get byte-alignment
    • A short CRC
    • Some additional flags

What did we do?

• Gave up guaranteed correct decompression
  – Some fields derived from link layer rather than copied from IP packet
• Ignores fate-sharing and end-to-end
  – It will almost be the same anyway
• Assumes constant session behavior
  – True for especially problematic sessions anyway!
What about performance then?

• Header compression *improves* performance
  – Time to compress, transfer and decompress is *shorter* than the time to transfer full header!

Really really fast IP lookups

• Problem: LPM is expensive

• Potential bottlenecks:
  – CPU due to complexity
  – Memory due to size of data structures
  – Data bus due to hardware design
What is expensive in lookups?

• Not CPU
  – Increasing speeds are easy

• Memory
  – Each access outside L2-cache is expensive

• Data bus
  – Data can only be forwarded that fast

Fast IP lookups: Basic idea

• Make data structure as small as possible
  – If possible, fit entirely in L1- or L2-cache
  – Minimize access to external memory

• Ignore the non-existing worst-case
  – Proper worst-case: All memory accesses to external memory
  – Realistic case: When reading from external memory, a cache line fills up

• Organize data structure to maximize usage of cache lines read from external memory!
Key idea 1: Organizing the data

• LPM matching is a dept-first lookup

• If a node match, we will continue down its subtree.
  – Put that subtree in the same cache line!

• Is this possible?!

Following the rules = a bad idea

• In algorithms and data structures class, we ask you to design a data structure that meets some criteria.
• Performance of data structures are typically depending on the \textit{density} of information they hold.
• Finding one data structure to rule them all when we want to optimize locality... nah!
Key idea 2: Hybrid data structures

- Store different subtrees of the search space in different data structures depending on the density of each subtree
- Optimize data structures for the given density
- Problem: What about when density changes?  
  - After all, IP routing is a dynamic system

Density changes in subtrees

- Density in a subtree increase slowly  
  - Adding an extra entry

- The process of adding an entry:
  - Lookup the location of the entry
    - Equivalent to making a LPM on the entry!
  - Put the entry beneath the found node
  - Cost vary depending on LPM complexity
Key idea 3: Prepare for change

- Make each insertion take equally long
  - Spend "left-over" cycles to prepare for change

- When the changed density cause a data structure transition, you are prepared
  - Transition will be cheap

Key idea 4: Constant-time service

- Make all lookups take equally long
- Amortize management costs over left-over cycles

- Main benefit: Synchronous process speed
  - Possible to tune memory and HW
  - Possible to implement really really fast
The key ideas to fast IP lookups

- Organize your data
  - Take memory model into account
- Hybrid data structures
  - There is no generally good data structure if performance is a real issue
- Prepare for transitions
  - It will make them smoother
- Amortize your costs
  - Possible to tune hardware

Fast IP lookups - performance

- 218k IPv4 entries in 2.7 Mb of memory
- Incremental updates 752 memory accesses in worst case scenario
- With 300 Mhz SRAM, we can forward at roughly 100 Gbit/s wire speeds, including 400,000 updates/second
- On a standard (old) machine, we can forward IPv4 at more than 10 Gbit/s
  - The bottleneck is now the data bus
  - On a modern computer: over 75 Gbit/s
Why bother about fast IP lookups?

- Not only IPv4 routing...
- IPv6
- Flow classification
- Firewalls
- IPsec devices
- ...

- What can you achieve with a 100 Mhz CPU?

IP in cellular devices

- Using IP header compression reduces overhead
  – 4-6 bytes/packet feasible

- Fast IP lookups enable full IPv6 stack in mobile

- Now, can we make it small enough?
YES WE CAN

TO BE CONTINUED...