COMP2221 Networks

David Head

University of Leeds

Lecture 5

Reminder of previous lecture

If the last few lectures we have covered the essential components of the Internet:

- Layered models of network architectures, focussing on the 5-layer TCP/IP model *(Lecture 2)*.
  - Application layer at the top.
  - ...
  - Physical layer at the bottom.
- **Ports**, which are assigned to processes rather than hosts *(Lecture 3)*.
- **DNS** (Domain Name System) servers, and how they map human-readable hostnames to IP addresses *(Lecture 4)*.

Todays lecture

Today's lecture is the first of 11 when we will look at actual network programming, in Java.

- The **Application** layer.

DNS queries (*i.e.* converting hostnames to IP address or *vice versa*) can be conveniently performed with `java.net.InetAddress`

- Will see a simple Java program that emulates `nslookup`.

First, it is necessary to look at IP addresses in more detail.

IP Addressing

Every **public** host on the internet has a unique IP address.

Two protocols are currently in use[1]:

- **IPv4**: Internet Protocol version 4.
- **IPv6**: Internet Protocol version 6.

IPv6 is gradually replacing IPv4, although there is no road map for the end of IPv4 (yet).

[1] IPv5 *was* in development, based on the Internet Stream Protocol ST-2, but after this was dropped IPv5 was never introduced for public use.

IPv4

Usually written as a 4-byte string of 4 integers.

- `a.b.c.d`
- Each byte takes a value from 0 to 255 (*i.e.* unsigned).
- The '4' in IPv4 does *not* refer to the number of bytes!

Some addresses and ranges of addresses have special meaning (see later).

Even without this, there are only $(256)^4 \approx 4.29 \times 10^9$ possible addresses, *i.e.* about 4.3 billion.

- The population of the Earth is currently about 8 billion.

Classful addressing

Originally there were several **classes** of IPv4 address:

- Class A: `0.*.*.*` to `127.*.*.*`
- Class B: `128.*.*.*` to `191.*.*.*`
- Class C: `192.*.*.*` to `223.*.*.*`
- Class D: `224.*.*.*` to `239.*.*.*`
- Class E: `240.*.*.*` to `255.*.*.*`

where the '*' means 'any value.'

They vary in size: Class A $>$ Class B $>$ Class C $>$ Class D, E.

Problems with the classful model

This so-called **classful** model is not perfect.

Various classes are reserved, reducing space.

- Class D: *Multi-casting (Lecture 15)*.
- Class E: Reserved for some unspecified 'future use.'
- Some special addresses:
  - `0.0.0.0` for 'this' machine.
  - `255.255.255.255` is used for **broadcasting** (device discovery).
  - `127.*.*.*` is used for *loopback*.

The least significant 3 bytes were not always managed efficiently.

Subnetworks in IPv4

In addition to their first byte ranges, the different classes also allows **subnetworks** to be defined:

- Class A allows 128 networks, each with $256^3 \approx 16.7$ million hosts: `a.*.*.*`
- Class B allows $64 \times 256 = 16384$ networks, each with 65,536 hosts: `a.b.*.*`
- Class C allows $32 \times 256^2 \approx 2$ million networks, each with 256 hosts: `a.b.c.*`

The idea was each organisation could choose a subnet.

- Most organisations need more than 256 hosts, but less than 65,536.
- Tended to select Class B and **under-utilise** their subnetwork.

CIDR: Classless Inter-Domain Routing

A first fix was to define subnetworks by *any* number of bits

- Greater range of subnetwork sizes (*i.e.* 256, 512, 1024, ...)
- Notation: `a.b.c.d/x` with `x` the number of common bits.

For example, 220.10.128.0/20 means 'all addresses that share their first 20 bits with 220.10.128.0':

- All of the first byte (220).
- All of the second byte (10).
- The most significant 4-bits of the third byte (128-143 inc.):
- Full range is:
  - 220.10.128.0
  - ...
  - 220.10.143.255

NAT: Network Address Translation

This allows more networks, and gaps in Class B networks can be filled, but there is still a limit on hosts.

A second fix was for **private** networks to have their own **internal** addresses, and only public-facing servers to has an actual IP address.

- Re-direct messages to/from private hosts using **ports**.
- `10.*.*.*` most common.
- Known as NAT (Network Address Translation), which we looked at in Lecture 4.

Not regarded as a permanent solution, more of a ‘quick fix.’

IPv6

The long term solution is to **expand the address space**.

- IPv6 uses 16-byte addresses.
- Total of $256^{16} \approx 3 \times 10^{38}$ possible addresses.
- Even if managed inefficiently, should never run out.

Currently around 45% users access Google *via* IPv6[1].

Some legacy systems do not support IPv6.

- Can wrap IPv6 datagrams into IPv4 datagrams if some intermediate routers only support IPv4.
- Known as **tunnelling** *(c.f. Lecture 17)*.

[1] `https://www.google.com/intl/en/ipv6/statistics.html`

IPv6 address format

Usually written in **hexadecimal**, with 8 **groups** *(pairs of bytes)* separated by colons. For example, `dns6.leeds.ac.uk` is[1]:

`2001:0630:0062:0059:0000:0000:0000:0053`

Can simplify by removing **leading zeros** in each group:

`2001:630:62:59:0:0:0:53`

Can further simplify by replacing **consecutive sections of zeros** by a double colon:

`2001:630:62:59::53`

[1] This *is* hexadecimal; there just happen to be no 'a's, 'b's *etc.*

Double double colons

Note you can only have **one double colon** in an address, otherwise it could be ambiguous.

For instance, the shortened address `2001:630::53::` could be `2001:0630:0000:0000:0053:0000:0000:0000`, or `2001:0630:0000:0000:0000:0053:0000:0000` *[check!]*

When there are multiple runs of all–zero values, only the **longest** should be contracted using a double colon.

- If the runs are the same length, the **leftmost** is contracted.

For example, `2001:0630:0000:0000:0053:01a0:0000:0000` is contracted to `2001:630::53:1a0:0:0`.

IPv6 subnetworks

CIDR also applies to IPv6, with the same slash notation '/'.

For instance,

`2001:630:62:59::/64`

specifies the range (not using double colons for clarity):

Start address	`2001:630:62:59:0:0:0:0`
End address	`2001:630:62:59:ffff:ffff:ffff:ffff`

The private address used to denote machines on the same local network can use the addresses:

- `10.*.*.*` or `10.0.0.0/8` in IPv4.
- `fc00::/8` in IPv6.

The `InetAddress` class

In Java, IP addresses are handled using the `InetAddress` class.

- Defined in `java.net`
- Used by most of the important network classes (including the ones we will use):
  - `Socket`
  - `ServerSocket`
  - `URL`
- Includes **both** the IP address **and** the host name.
- Used to represent **both** IPv4 **and** IPv6 addresses.

Making an `InetAddress` object (1)

There is no public constructor for `InetAddress`. Instead, three **static** methods can return an `InetAddress` object:

```
public static InetAddress getByName( String hostname )
```

- Most common.

```
public static InetAddress[] getAllByName( String hostname )
```

- Returns an array of `InetAddress` objects.
- Depends on the configuration of the local DNS server.

```
public static InetAddress getLocalHost()
```

- *i.e.* the address of the host running the code.

Making an `InetAddress` object (2)

These methods do not simply set the object state by using the constructor arguments (which is how a public constructor usually works).

- They make **DNS queries** on your behalf.

Do not need to specify IPv4 or IPv6.

- Uses **polymorphism**.
- Has subclasses `Inet4Address` and `Inet6Address`.
- User is usually oblivious to which subclass has been returned.
- Can specify the IPv4 or IPv6 subclass if desired.

Factory Methods for `InetAddress`

Creating objects in this way, without specifying the exact class to be created, is called the **factory method pattern** in OO programming.

Here it means that:

- We supply a string that **may** resolve to an address.
- We do not know in advance if the hostname exists.
- The factory method returns either an `Inet4Address` or an `Inet6Address` object.
- Through polymorphism, we can refer to either as `InetAddress`.

Issues?

These methods will connect to a DNS server.

- If you are not connected, it may prompt for a connection, or throw an exception (see below).

Make their **own external network connections** to get the information they need:

- Not normal constructor behaviour.
- Can fail for a variety of reasons (no network connection, unknown host, security *etc.*)
- Need to catch `UnknownHostException` (checked exception).
- Relatively expensive (slow), due to the DNS overhead (although will use DNS cache where possible).

Getters

Some common 'get' methods to access an instance of `InetAddress` are:

`public String getHostName()`

- Returns the hostname you used to create the object.

`public String getHostAddress()`

- Textual representation of the IP address (IPv4 or IPv6).

`public byte[] getAddress()`

- Returns the IP address as an array of 4 or 16 bytes.

In addition, `System.out.println()` will print a combination of hostname and IP address.

Setters

. . . there are **no** public 'set' methods for `InetAddress`!

- It is an **immutable** object whose state cannot be changed once created.
- At least, not externally - internal state changes are possible with private methods.

Why is this?

- We assume DNS sets the **correct** state.
- Letting the user change parts of the object runs the risk of breaking consistency with DNS.

OO Design

Encapsulation

- Public access to getters.
- Private constructors and setters.

Inheritance

- `Inet4Address` and `Inet6Address extend InetAddress`.

Polymorphism

- Typically use the parent `InetAddress` class.
- All we need in the Application layer.

Abstraction

- We do not need to know if our address is IPv4 or IPv6, only that it resolves to a host that exists.

Some other methods

`public boolean isReachable( int timeOut )`

- Tests if the address is reachable.
- Waits a maximum of `timeOut` milliseconds.
- Similar to `ping`, but uses IP rather than ICMP *[cf. Lecture 17]*.

`public boolean isLoopBackAddress()`

- Loopback addresses are returned as soon as they reach the Network layer.
- Useful for testing without physical infrastructure.
- IPv4: `127.*.*.*`, or `127.0.0.0/8`
- IPv6: `0:0:0:0:0:0:0:1`, usually written `::1`.

Summary and next time

Today we have seen

- How IPv4 and IPv6 addresses are structured.
- How to create instances of `java.net.InetAddress`.

Next time we will see a simple application of `InetAddress`, and look at Java I/O streams, essential for sending data to, or receiving data from, the internet.