Concurrent Computing (Computer Networks)

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨Daniel.Page@bristol.ac.uk⟩

March 14, 2016

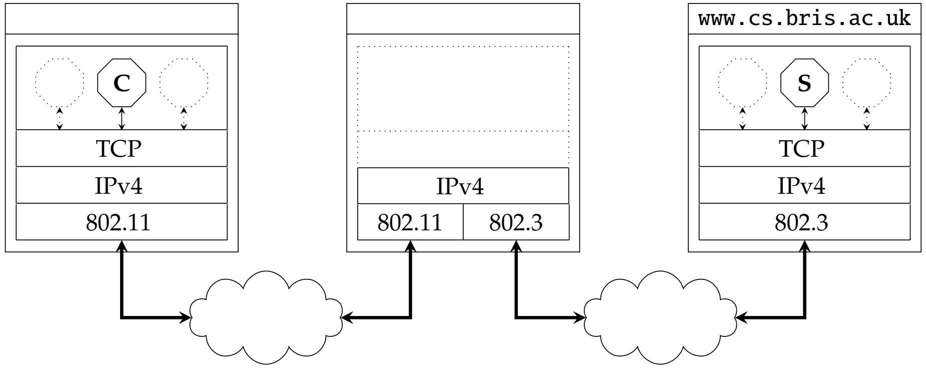
Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

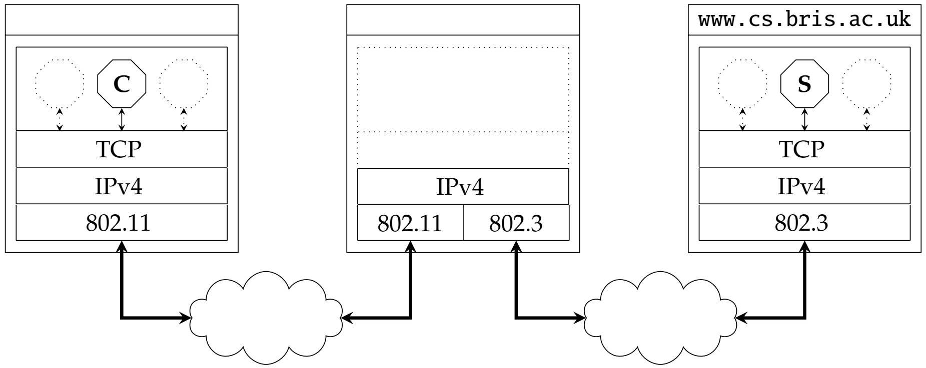
- Recall: we know how to realise



st. hosts can transmit TCP segments to each other.

Notes:

- Recall: we know how to realise



st. hosts can transmit TCP segments to each other ...

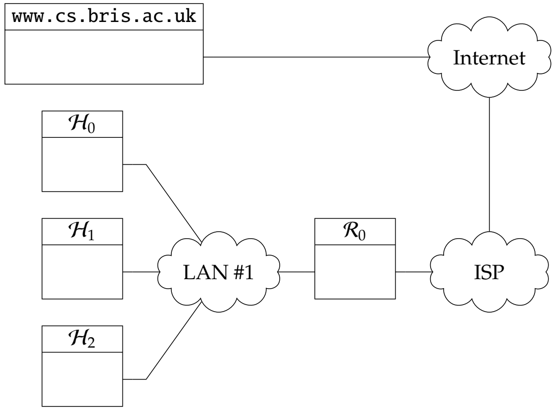
- ... *but*

1. how *could* both LANs use the same private IP address block, and
2. how does the client know the IP address of `www.cs.bris.ac.uk`?

Notes:

Problem #1 $\rightsquigarrow$ NAT (1)

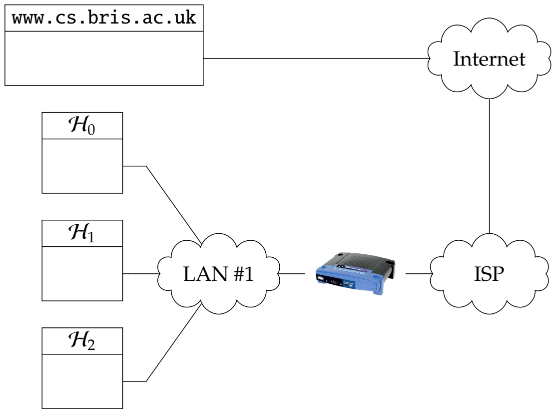
- Problem: consider the following inter-network



Notes:

Problem #1 $\rightsquigarrow$ NAT (1)

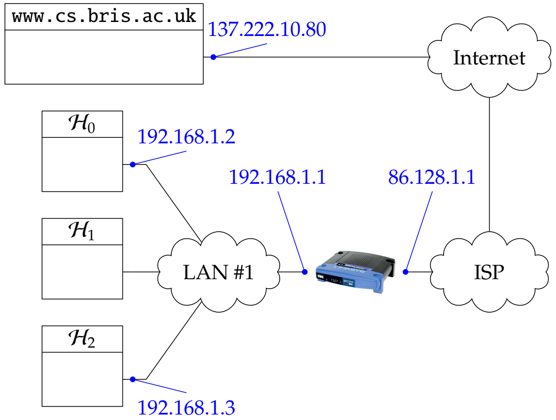
- Problem: consider the following inter-network



Notes:

Problem #1 $\rightsquigarrow$ NAT (1)

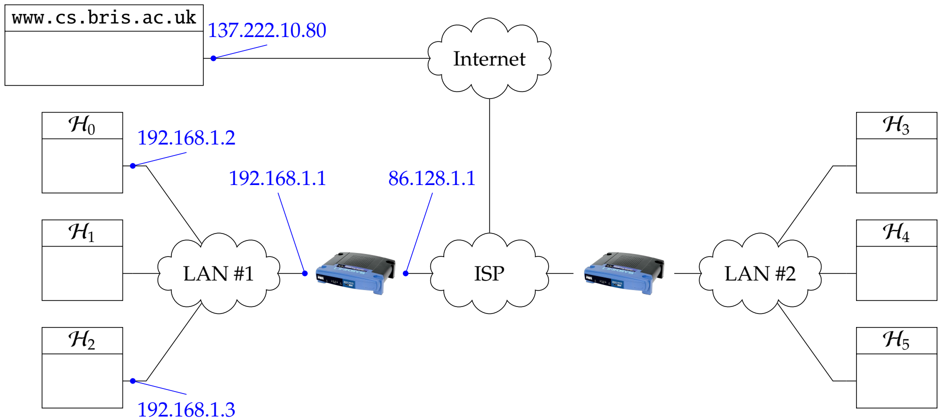
- Problem: consider the following inter-network



Notes:

Problem #1 $\rightsquigarrow$ NAT (1)

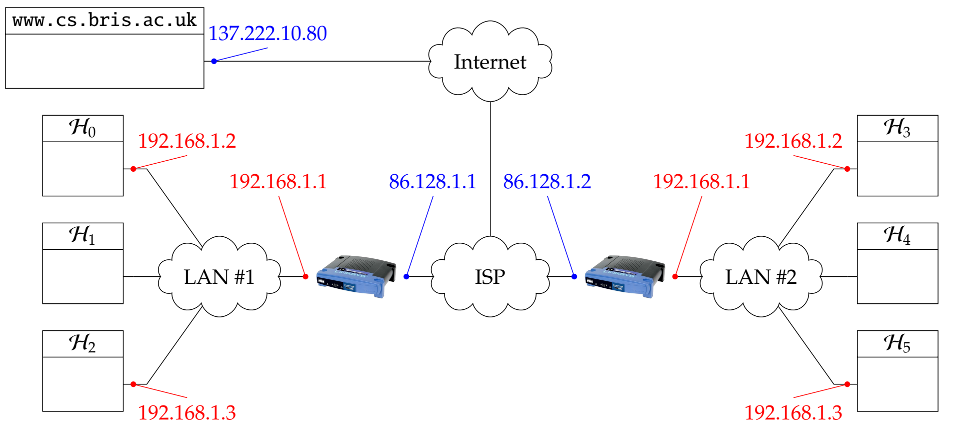
- Problem: consider the following inter-network



Notes:

Problem #1 ⇝ NAT (1)

- Problem: consider the following inter-network

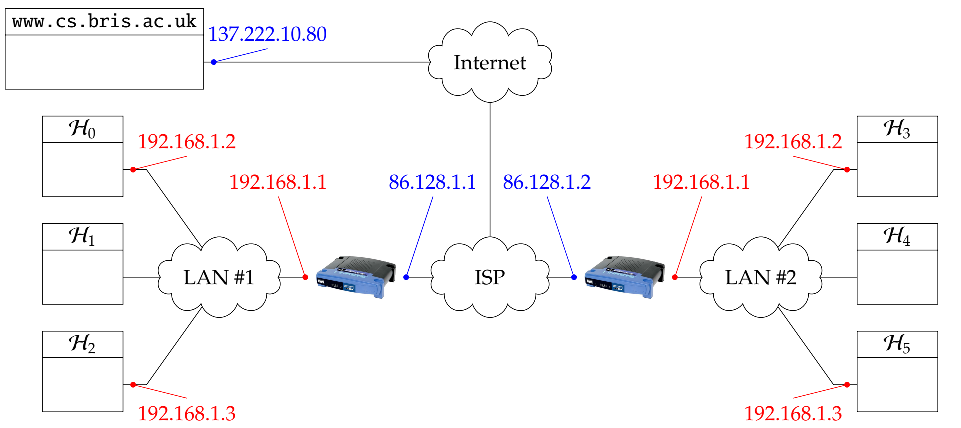


where LANs #1 *and* #2 (legitimately) use the private address block 192.168.0.0/16, so the host IP addresses conflict.

Notes:

Problem #1 ⇝ NAT (1)

- Problem: consider the following inter-network

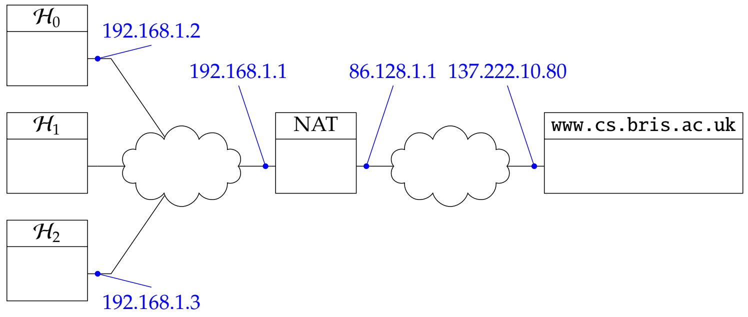


where LANs #1 *and* #2 (legitimately) use the private address block 192.168.0.0/16, so the host IP addresses conflict.

- (A) solution: **Network Address Translation (NAT)** [8].

Notes:

- Example:



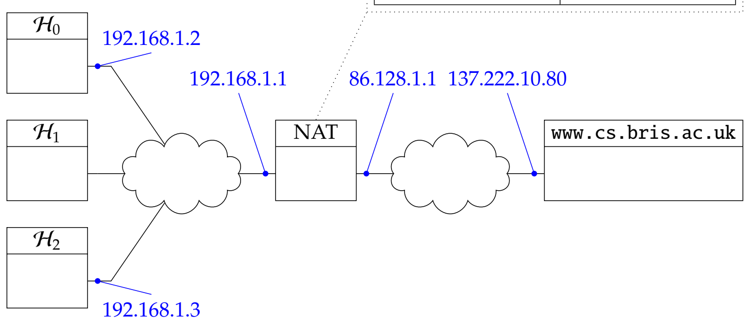
Notes:

-
- This specific realisation of NAT is actually termed **IP masquerading**: it could be thought of as multiplexing n external IP addresses between m internal hosts (who have m internal IP addresses).
  Either way, the point is that NAT is, strictly speaking, a more general technique that can manipulate addresses for whatever purpose makes sense: the name is abused a little, since arguably the most common use-case for NAT is as described.
- Although there's no requirement that $n = 1$ as it is here, it is normal to find $m > n$: if not, that implies there are enough external IP addresses for each host anyway (although managing them dynamically might still mean NAT is attractive).
  This highlights the fact that NAT really performs address *and* port translation; the port aspect is needed for disambiguation, otherwise we'd be unable to know which internal host an inbound packet was for (because the external IP address is shared between multiple such hosts). You may see it called **Port Address Translation (PAT)** to emphasis this fact.
- Where the NAT appliance is also acting as a router, you could argue the overhead of translation is limited in the sense it needs to forward any packets anyway: the look-up and translation process *is* overhead, but just adds some fraction to what is already a strong efficiency requirement.

Problem #1 ⇝ NAT (2)

- Example:

Internal		External	
Address	Port	Address	Port
192.168.1.3	1234	86.128.1.1	4923
192.168.1.3	5678	86.128.1.1	3811
192.168.1.2	1234	86.128.1.1	3947
⋮	⋮	⋮	⋮



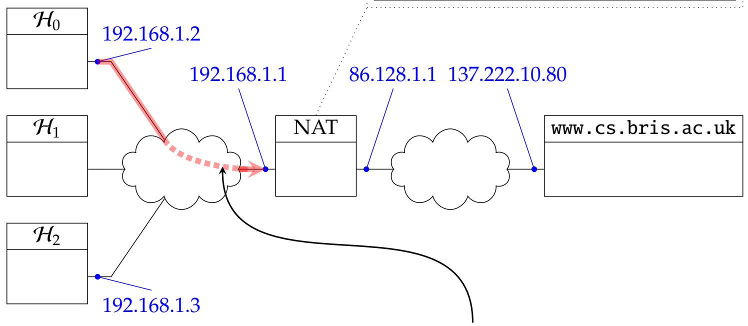
Notes:

-
- This specific realisation of NAT is actually termed **IP masquerading**: it could be thought of as multiplexing n external IP addresses between m internal hosts (who have m internal IP addresses).
  Either way, the point is that NAT is, strictly speaking, a more general technique that can manipulate addresses for whatever purpose makes sense: the name is abused a little, since arguably the most common use-case for NAT is as described.
- Although there's no requirement that $n = 1$ as it is here, it is normal to find $m > n$: if not, that implies there are enough external IP addresses for each host anyway (although managing them dynamically might still mean NAT is attractive).
  This highlights the fact that NAT really performs address *and* port translation; the port aspect is needed for disambiguation, otherwise we'd be unable to know which internal host an inbound packet was for (because the external IP address is shared between multiple such hosts). You may see it called **Port Address Translation (PAT)** to emphasis this fact.
- Where the NAT appliance is also acting as a router, you could argue the overhead of translation is limited in the sense it needs to forward any packets anyway: the look-up and translation process *is* overhead, but just adds some fraction to what is already a strong efficiency requirement.

Problem #1 ⇝ NAT (2)

- Example:

Internal Address	Internal Port	External Address	External Port
192.168.1.3	1234	86.128.1.1	4923
192.168.1.3	5678	86.128.1.1	3811
192.168.1.2	1234	86.128.1.1	3947
⋮	⋮	⋮	⋮



H_{IPv4}[src = 192.168.1.2, dst = 137.222.10.80] $\parallel$ H_{TCP}[syn = **true**, src port = 5678, dst port = 8(

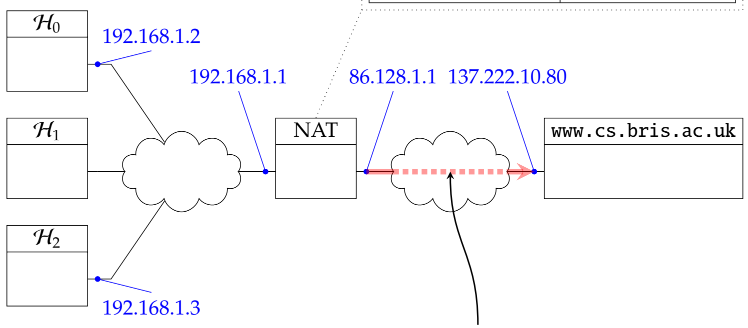
Notes:

-
- This specific realisation of NAT is actually termed **IP masquerading**: it could be thought of as multiplexing n external IP addresses between m internal hosts (who have m internal IP addresses).
  Either way, the point is that NAT is, strictly speaking, a more general technique that can manipulate addresses for whatever purpose makes sense: the name is abused a little, since arguably the most common use-case for NAT is as described.
- Although there's no requirement that $n = 1$ as it is here, it is normal to find $m > n$: if not, that implies there are enough external IP addresses for each host anyway (although managing them dynamically might still mean NAT is attractive).
  This highlights the fact that NAT really performs address *and* port translation; the port aspect is needed for disambiguation, otherwise we'd be unable to know which internal host an inbound packet was for (because the external IP address is shared between multiple such hosts). You may see it called **Port Address Translation (PAT)** to emphasis this fact.
- Where the NAT appliance is also acting as a router, you could argue the overhead of translation is limited in the sense it needs to forward any packets anyway: the look-up and translation process *is* overhead, but just adds some fraction to what is already a strong efficiency requirement.

Problem #1 ⇝ NAT (2)

- Example:

Internal Address	Internal Port	External Address	External Port
192.168.1.3	1234	86.128.1.1	4923
192.168.1.3	5678	86.128.1.1	3811
192.168.1.2	1234	86.128.1.1	3947
192.168.1.2	5678	86.128.1.1	4853
⋮	⋮	⋮	⋮



H_{IPv4}[src = 86.120.1.1, dst = 137.222.10.80] $\parallel$ H_{TCP}[syn = **true**, src port = 4853, dst port = 8(

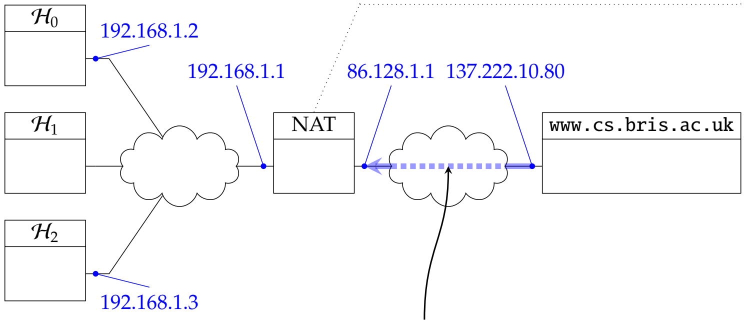
Notes:

-
- This specific realisation of NAT is actually termed **IP masquerading**: it could be thought of as multiplexing n external IP addresses between m internal hosts (who have m internal IP addresses).
  Either way, the point is that NAT is, strictly speaking, a more general technique that can manipulate addresses for whatever purpose makes sense: the name is abused a little, since arguably the most common use-case for NAT is as described.
- Although there's no requirement that $n = 1$ as it is here, it is normal to find $m > n$: if not, that implies there are enough external IP addresses for each host anyway (although managing them dynamically might still mean NAT is attractive).
  This highlights the fact that NAT really performs address *and* port translation; the port aspect is needed for disambiguation, otherwise we'd be unable to know which internal host an inbound packet was for (because the external IP address is shared between multiple such hosts). You may see it called **Port Address Translation (PAT)** to emphasis this fact.
- Where the NAT appliance is also acting as a router, you could argue the overhead of translation is limited in the sense it needs to forward any packets anyway: the look-up and translation process *is* overhead, but just adds some fraction to what is already a strong efficiency requirement.

Problem #1 ⇝ NAT (2)

- Example:

Internal		External	
Address	Port	Address	Port
192.168.1.3	1234	86.128.1.1	4923
192.168.1.3	5678	86.128.1.1	3811
192.168.1.2	1234	86.128.1.1	3947
192.168.1.2	5678	86.128.1.1	4853
⋮	⋮	⋮	⋮



H_{IPv4}[src = 137.222.10.80, dst = 86.120.1.1] ∥ H_{TCP}[syn = **true**, ack = **true**, src port = 80, dst port = 485

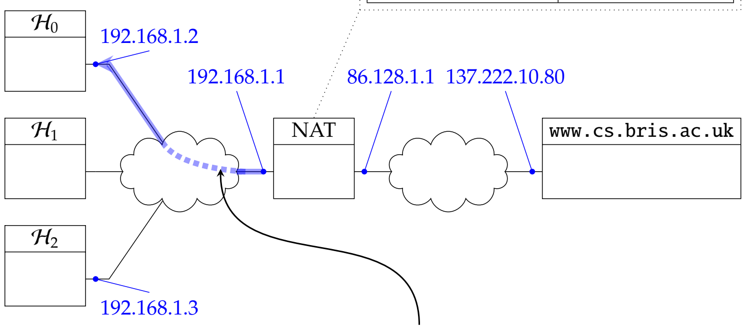
Notes:

-
- This specific realisation of NAT is actually termed **IP masquerading**: it could be thought of as multiplexing n external IP addresses between m internal hosts (who have m internal IP addresses).
  Either way, the point is that NAT is, strictly speaking, a more general technique that can manipulate addresses for whatever purpose makes sense: the name is abused a little, since arguably the most common use-case for NAT is as described.
- Although there's no requirement that $n = 1$ as it is here, it is normal to find $m > n$: if not, that implies there are enough external IP addresses for each host anyway (although managing them dynamically might still mean NAT is attractive).
  This highlights the fact that NAT really performs address *and* port translation; the port aspect is needed for disambiguation, otherwise we'd be unable to know which internal host an inbound packet was for (because the external IP address is shared between multiple such hosts). You may see it called **Port Address Translation (PAT)** to emphasis this fact.
- Where the NAT appliance is also acting as a router, you could argue the overhead of translation is limited in the sense it needs to forward any packets anyway: the look-up and translation process *is* overhead, but just adds some fraction to what is already a strong efficiency requirement.

Problem #1 ⇝ NAT (2)

- Example:

Internal		External	
Address	Port	Address	Port
192.168.1.3	1234	86.128.1.1	4923
192.168.1.3	5678	86.128.1.1	3811
192.168.1.2	1234	86.128.1.1	3947
192.168.1.2	5678	86.128.1.1	4853
⋮	⋮	⋮	⋮



H_{IPv4}[src = 137.222.10.80, dst = 192.168.1.2] ∥ H_{TCP}[syn = **true**, ack = **true**, src port = 80, dst port = 567

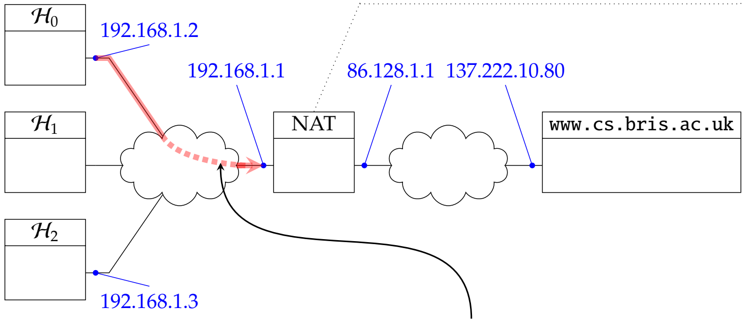
Notes:

-
- This specific realisation of NAT is actually termed **IP masquerading**: it could be thought of as multiplexing n external IP addresses between m internal hosts (who have m internal IP addresses).
  Either way, the point is that NAT is, strictly speaking, a more general technique that can manipulate addresses for whatever purpose makes sense: the name is abused a little, since arguably the most common use-case for NAT is as described.
- Although there's no requirement that $n = 1$ as it is here, it is normal to find $m > n$: if not, that implies there are enough external IP addresses for each host anyway (although managing them dynamically might still mean NAT is attractive).
  This highlights the fact that NAT really performs address *and* port translation; the port aspect is needed for disambiguation, otherwise we'd be unable to know which internal host an inbound packet was for (because the external IP address is shared between multiple such hosts). You may see it called **Port Address Translation (PAT)** to emphasis this fact.
- Where the NAT appliance is also acting as a router, you could argue the overhead of translation is limited in the sense it needs to forward any packets anyway: the look-up and translation process *is* overhead, but just adds some fraction to what is already a strong efficiency requirement.

Problem #1 ⇝ NAT (2)

- Example:

Internal Address	Internal Port	External Address	External Port
192.168.1.3	1234	86.128.1.1	4923
192.168.1.3	5678	86.128.1.1	3811
192.168.1.2	1234	86.128.1.1	3947
192.168.1.2	5678	86.128.1.1	4853
⋮	⋮	⋮	⋮



H_{IPv4}[src = 192.168.1.2, dst = 137.222.10.80] || H_{TCP}[ack = **true**, src port = 5678, dst port = 8(

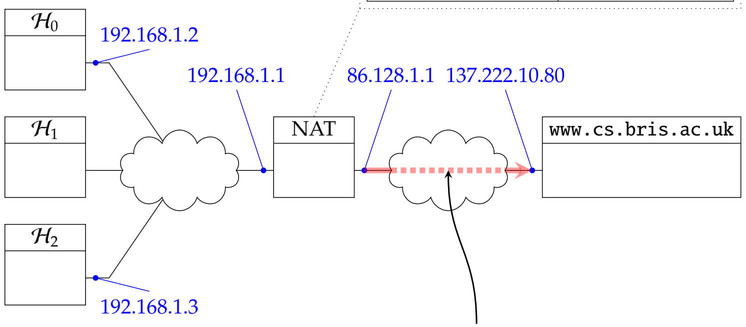
Notes:

-
- This specific realisation of NAT is actually termed **IP masquerading**: it could be thought of as multiplexing n external IP addresses between m internal hosts (who have m internal IP addresses).
  Either way, the point is that NAT is, strictly speaking, a more general technique that can manipulate addresses for whatever purpose makes sense: the name is abused a little, since arguably the most common use-case for NAT is as described.
- Although there's no requirement that $n = 1$ as it is here, it is normal to find $m > n$: if not, that implies there are enough external IP addresses for each host anyway (although managing them dynamically might still mean NAT is attractive).
  This highlights the fact that NAT really performs address *and* port translation; the port aspect is needed for disambiguation, otherwise we'd be unable to know which internal host an inbound packet was for (because the external IP address is shared between multiple such hosts). You may see it called **Port Address Translation (PAT)** to emphasis this fact.
- Where the NAT appliance is also acting as a router, you could argue the overhead of translation is limited in the sense it needs to forward any packets anyway: the look-up and translation process *is* overhead, but just adds some fraction to what is already a strong efficiency requirement.

Problem #1 ⇝ NAT (2)

- Example:

Internal Address	Internal Port	External Address	External Port
192.168.1.3	1234	86.128.1.1	4923
192.168.1.3	5678	86.128.1.1	3811
192.168.1.2	1234	86.128.1.1	3947
192.168.1.2	5678	86.128.1.1	4853
⋮	⋮	⋮	⋮



H_{IPv4}[src = 86.120.1.1, dst = 137.222.10.80] || H_{TCP}[ack = **true**, src port = 4853, dst port = 8(

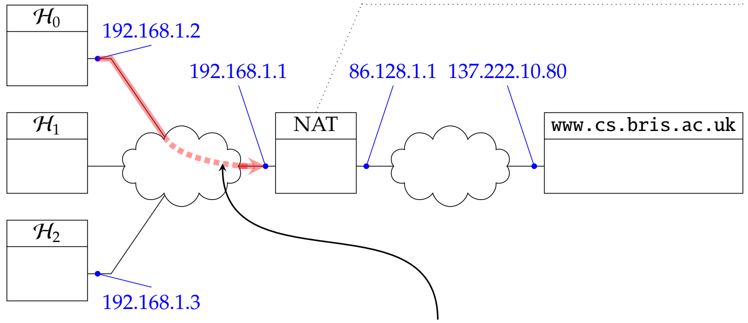
Notes:

-
- This specific realisation of NAT is actually termed **IP masquerading**: it could be thought of as multiplexing n external IP addresses between m internal hosts (who have m internal IP addresses).
  Either way, the point is that NAT is, strictly speaking, a more general technique that can manipulate addresses for whatever purpose makes sense: the name is abused a little, since arguably the most common use-case for NAT is as described.
- Although there's no requirement that $n = 1$ as it is here, it is normal to find $m > n$: if not, that implies there are enough external IP addresses for each host anyway (although managing them dynamically might still mean NAT is attractive).
  This highlights the fact that NAT really performs address *and* port translation; the port aspect is needed for disambiguation, otherwise we'd be unable to know which internal host an inbound packet was for (because the external IP address is shared between multiple such hosts). You may see it called **Port Address Translation (PAT)** to emphasis this fact.
- Where the NAT appliance is also acting as a router, you could argue the overhead of translation is limited in the sense it needs to forward any packets anyway: the look-up and translation process *is* overhead, but just adds some fraction to what is already a strong efficiency requirement.

Problem #1 ⇝ NAT (2)

- Example:

Internal		External	
Address	Port	Address	Port
192.168.1.3	1234	86.128.1.1	4923
192.168.1.3	5678	86.128.1.1	3811
192.168.1.2	1234	86.128.1.1	3947
192.168.1.2	5678	86.128.1.1	4853
⋮	⋮	⋮	⋮



$H_{IPv4}[\text{src} = 192.168.1.2, \text{dst} = 137.222.10.80] \parallel H_{TCP}[\text{syn} = \textbf{true}, \text{src port} = 5678, \text{dst port} = 80] \parallel \langle ..$

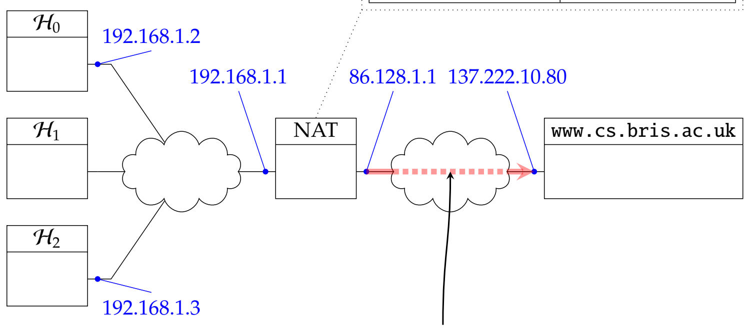
Notes:

-
- This specific realisation of NAT is actually termed **IP masquerading**: it could be thought of as multiplexing n external IP addresses between m internal hosts (who have m internal IP addresses).
  Either way, the point is that NAT is, strictly speaking, a more general technique that can manipulate addresses for whatever purpose makes sense: the name is abused a little, since arguably the most common use-case for NAT is as described.
- Although there's no requirement that $n = 1$ as it is here, it is normal to find $m > n$: if not, that implies there are enough external IP addresses for each host anyway (although managing them dynamically might still mean NAT is attractive).
  This highlights the fact that NAT really performs address *and* port translation; the port aspect is needed for disambiguation, otherwise we'd be unable to know which internal host an inbound packet was for (because the external IP address is shared between multiple such hosts). You may see it called **Port Address Translation (PAT)** to emphasis this fact.
- Where the NAT appliance is also acting as a router, you could argue the overhead of translation is limited in the sense it needs to forward any packets anyway: the look-up and translation process *is* overhead, but just adds some fraction to what is already a strong efficiency requirement.

Problem #1 ⇝ NAT (2)

- Example:

Internal		External	
Address	Port	Address	Port
192.168.1.3	1234	86.128.1.1	4923
192.168.1.3	5678	86.128.1.1	3811
192.168.1.2	1234	86.128.1.1	3947
192.168.1.2	5678	86.128.1.1	4853
⋮	⋮	⋮	⋮



$H_{IPv4}[\text{src} = 86.120.1.1, \text{dst} = 137.222.10.80] \parallel H_{TCP}[\text{syn} = \textbf{true}, \text{src port} = 4853, \text{dst port} = 80] \parallel \langle ..$

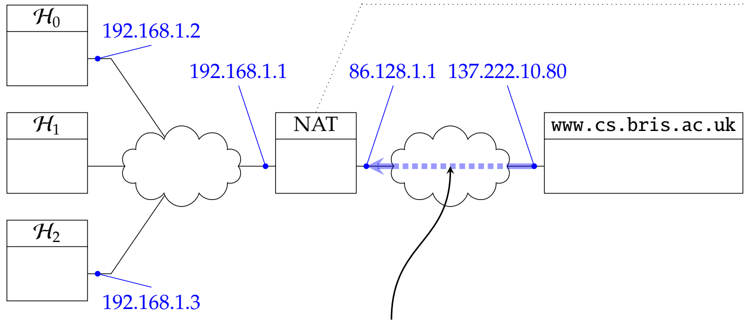
Notes:

-
- This specific realisation of NAT is actually termed **IP masquerading**: it could be thought of as multiplexing n external IP addresses between m internal hosts (who have m internal IP addresses).
  Either way, the point is that NAT is, strictly speaking, a more general technique that can manipulate addresses for whatever purpose makes sense: the name is abused a little, since arguably the most common use-case for NAT is as described.
- Although there's no requirement that $n = 1$ as it is here, it is normal to find $m > n$: if not, that implies there are enough external IP addresses for each host anyway (although managing them dynamically might still mean NAT is attractive).
  This highlights the fact that NAT really performs address *and* port translation; the port aspect is needed for disambiguation, otherwise we'd be unable to know which internal host an inbound packet was for (because the external IP address is shared between multiple such hosts). You may see it called **Port Address Translation (PAT)** to emphasis this fact.
- Where the NAT appliance is also acting as a router, you could argue the overhead of translation is limited in the sense it needs to forward any packets anyway: the look-up and translation process *is* overhead, but just adds some fraction to what is already a strong efficiency requirement.

Problem #1 ⇝ NAT (2)

- Example:

Internal		External	
Address	Port	Address	Port
192.168.1.3	1234	86.128.1.1	4923
192.168.1.3	5678	86.128.1.1	3811
192.168.1.2	1234	86.128.1.1	3947
192.168.1.2	5678	86.128.1.1	4853
⋮	⋮	⋮	⋮



$H_{IPv4}[\text{src} = 137.222.10.80, \text{dst} = 86.120.1.1] \parallel H_{TCP}[\text{syn} = \textbf{true}, \text{ack} = \textbf{true}, \text{src port} = 80, \text{dst port} = 4853] \parallel \langle ..$

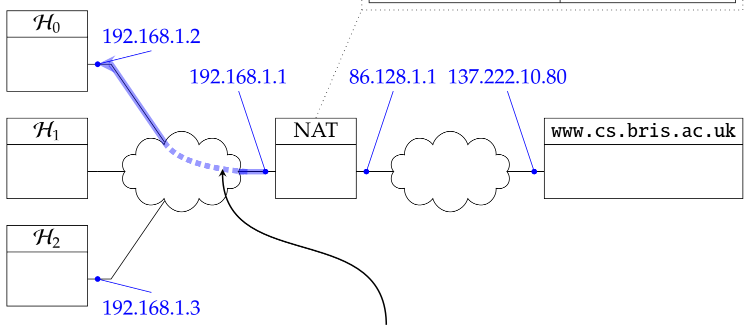
Notes:

-
- This specific realisation of NAT is actually termed **IP masquerading**: it could be thought of as multiplexing n external IP addresses between m internal hosts (who have m internal IP addresses).
  Either way, the point is that NAT is, strictly speaking, a more general technique that can manipulate addresses for whatever purpose makes sense: the name is abused a little, since arguably the most common use-case for NAT is as described.
- Although there's no requirement that $n = 1$ as it is here, it is normal to find $m > n$: if not, that implies there are enough external IP addresses for each host anyway (although managing them dynamically might still mean NAT is attractive).
  This highlights the fact that NAT really performs address *and* port translation; the port aspect is needed for disambiguation, otherwise we'd be unable to know which internal host an inbound packet was for (because the external IP address is shared between multiple such hosts). You may see it called **Port Address Translation (PAT)** to emphasis this fact.
- Where the NAT appliance is also acting as a router, you could argue the overhead of translation is limited in the sense it needs to forward any packets anyway: the look-up and translation process *is* overhead, but just adds some fraction to what is already a strong efficiency requirement.

Problem #1 ⇝ NAT (2)

- Example:

Internal		External	
Address	Port	Address	Port
192.168.1.3	1234	86.128.1.1	4923
192.168.1.3	5678	86.128.1.1	3811
192.168.1.2	1234	86.128.1.1	3947
192.168.1.2	5678	86.128.1.1	4853
⋮	⋮	⋮	⋮



$H_{IPv4}[\text{src} = 137.222.10.80, \text{dst} = 192.168.1.2] \parallel H_{TCP}[\text{syn} = \textbf{true}, \text{ack} = \textbf{true}, \text{src port} = 80, \text{dst port} = 5678] \parallel \langle ..$

Notes:

-
- This specific realisation of NAT is actually termed **IP masquerading**: it could be thought of as multiplexing n external IP addresses between m internal hosts (who have m internal IP addresses).
  Either way, the point is that NAT is, strictly speaking, a more general technique that can manipulate addresses for whatever purpose makes sense: the name is abused a little, since arguably the most common use-case for NAT is as described.
- Although there's no requirement that $n = 1$ as it is here, it is normal to find $m > n$: if not, that implies there are enough external IP addresses for each host anyway (although managing them dynamically might still mean NAT is attractive).
  This highlights the fact that NAT really performs address *and* port translation; the port aspect is needed for disambiguation, otherwise we'd be unable to know which internal host an inbound packet was for (because the external IP address is shared between multiple such hosts). You may see it called **Port Address Translation (PAT)** to emphasis this fact.
- Where the NAT appliance is also acting as a router, you could argue the overhead of translation is limited in the sense it needs to forward any packets anyway: the look-up and translation process *is* overhead, but just adds some fraction to what is already a strong efficiency requirement.

Problem #1 ⇝ NAT (3)

- NAT appliances are examples of **middlebox** [6]:
  - exist "in" network (like routers), but
  - operate in more than network layer (i.e., don't *just* forward packets, like hosts), so
  - break layered model and connectivity, and can result in strange side-effects

  hence, NAT specifically
  - Good:
    - helps mitigate IPv4 address scarcity problem, and
    - can be easily, centrally deployed
  - Bad:
    - is difficult to extend beyond TCP,
    - requires initial outgoing connection, and
    - potentially fails if application layer depends on and/or exposes (internal) IP address

  plus, internal hosts are anonymised to some extent since their traffic flows are merged externally.

Notes:

- Some of disadvantages can be mitigated to some extent; **port forwarding** can, for example, be used to expose specific ports of an internal host to the external network.

Problem #2 ⇝ DNS (1)

- Problem:

Definition (name, address and resolution)

- a (human-readable) **name** is used to *identify* the resource,
- an (machine-readable) **address** is used to *locate* the resource, and
- **resolution** (resp. **reverse resolution**) maps a name to an address (resp. address to a name).

so, we need some mechanism to perform

$$\texttt{www.cs.bris.ac.uk} \underset{\text{reverse resolve}}{\overset{\text{resolve}}{\rightleftarrows}} 137.222.10.80$$

where

- the LHS is a **domain name**, and
- the RHS is an IP address.

Notes:

- A good analogy is the telephone directory, which maps names (of people) to addresses (or more specifically, their physical address and/or their telephone number); in printed form at least, there is normally no way to perform the reverse. As such, resolution (resp. reverse resolution) obviously just a formal version of the more common term look-up (resp. reverse look-up).
- The mapping between names and addresses need not *necessarily* be one-to-one: it's obviously possible for a web-server to have more than one IP address for example, in the same way any IP-connected host can, and to house more than one web-site (i.e., for that IP address to map from several URLs).
- There's a beautifully constructed, animated introduction to DNS at

  `http://howdns.works/`

- Solutions:
  1. fully centralised: pre ~ 1984 via ARPANET `hosts.txt` [9], e.g.,

     ```
     NET : 128.54.0.0 : UCSD :
     ```

     and

     ```
     HOST : 128.54.0.1 : SDCSVAX,UCSD : VAX-11/780 : UNIX : TCP/TELNET,TCP/FTP,TCP/SMTP,UDP :
     ```

  2. partially decentralised: via
     - in `/etc/networks`, e.g.,

       ```
       loopback 127.0.0.0
       ```

       and
     - in `/etc/hosts`, e.g.,

       ```
       127.0.0.1 localhost
       ```

     on a POSIX-style OS (e.g., Linux),
  3. fully decentralised: post ~ 1984 via **Domain Name System (DNS)** [11, 12] which includes
     3.1 a managed, hierarchical **name space**,
     3.2 a protocol, used to communicate queries and responses, plus
     3.3 an infrastructure, comprised of **name servers**, used to host the (distributed) database and respond to queries.

Notes:

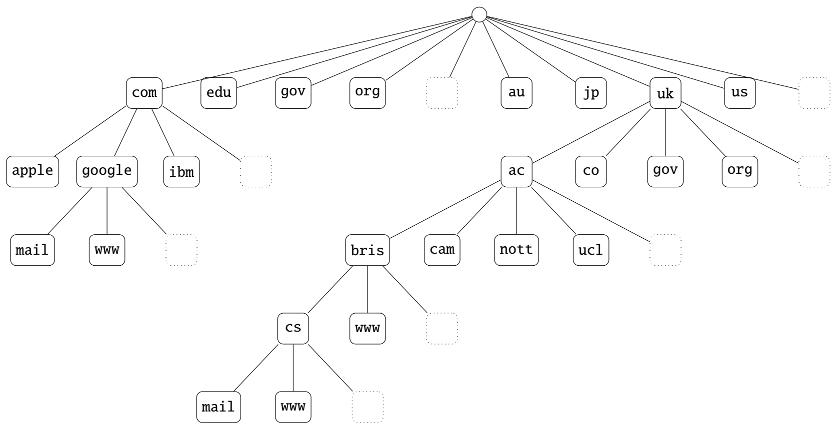
- You can still download an archived copy of the original ARPANET `hosts.txt` file; see, for example,

  http://jim.rees.org/apollo-archive/hosts.txt

  Originally this file was maintained and hosted by the Stanford Research Institute (SRI).
- Hopefully it's clear that the centralised `hosts.txt` solution isn't
  - efficient (e.g., a host may only need one mapping but needs to download the entire database, there's a delay wrt. propagation of changes), or
  - scalable (e.g., the database of mappings is potentially *very* large, and the server is potentially placed under high load)

  Likewise the partially decentralised solution is useful in some contexts but isn't really automated: `/etc/hosts` probably needs to be configured manually (at some point).
- Note that
  1. the DNS protocol is UDP-based; it operates via port 53 (bar situations where the message length is prohibitive, when it falls-back to using TCP), and implements ARQ for reliability, and
  2. the DNS database is somewhat general-purpose, allowing various record types, e.g.,

Type	Description
SOA	an authority record
NS	a name server record
A	an address record
CNAME	a canonical name record
MX	a mail exchange record
TXT	a text record



Notes:

- ICANN manages assignment of and infrastructure for TLDs, and likewise for the root zone (i.e., the zone for the root of the name space); it does so as a sort of commercial stand-in for IANA.
- IANA maintain a definitive set of reserved TLDs [7] at

  http://www.iana.org/assignments/special-use-domain-names

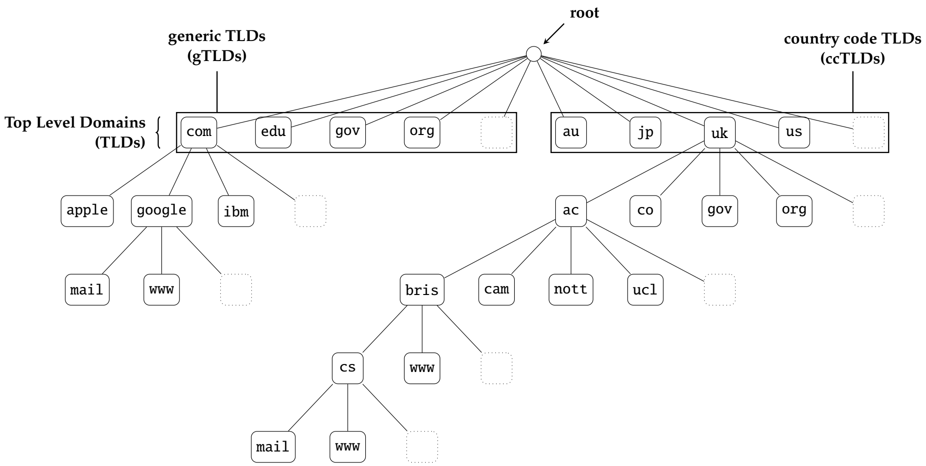
  which include examples such as `arpa`, which was originally used to transition the ARPANET to the Internet (i.e., all ARPANET names were initially "ported" into this part of the DNS name space).
- One can distinguish between a **Fully Qualified Domain Name (FQDN)** and **Partially Qualified Domain Name (PQDN)**: the former includes all components between root and host, whereas the latter does not (e.g., is relative to some base name). This concept is basically the same as in other (hierarchical) naming systems, e.g., a UNIX file system where absolute and relative paths are analogies.
- The majority of the `uk` TLD is managed by Nominet

  http://www.nominet.org.uk/

  with (some) exceptions including `gov.uk` and `ac.uk`, which is managed by JISC

  http://www.jisc.ac.uk/

- To (fully) specify the name of a leaf node, we simply traverse the name space tree toward the root adding a dot between the names at each level. The root node name is the empty string, meaning that *formally* the name produced will end with `.`; informally, we would typically omit this.
- DNS supports wildcard names, e.g., `*.cs.bris.ac.uk`; see [10] for a detailed overview.
- A DNS zone is simply a contiguous portion of the DNS name space, i.e., sub-tree of the hierarchy.
- Division of the name space into zones aims to offer scalability. The aim is to form a covering set of the entire name space: by having queries relating to a given zone managed by a dedicated name server, this allows a) local management of the name space in that zone, b) provision of the infrastructure required to handle queries for that zone.
- In a sense, the approach of zoning in DNS is is somewhat similar to the use of subnetting with IP addresses: in both cases, the hierarchy imposed offers some help wrt. scalability. It's important to point out differences, however. For example, the DNS hierarchy could be described as logical in the sense there's no real reason `x.example.com` and `y.example.com` are connected in any sense than the name space; with an IP sub-net however, there must be a physical relationship of sorts because the approach to routing depends on it to some extent.
- In delegating from one zone to another (sub-)zone, the former basically acts as a pointer to the latter: if the zone name server is presented with a query too specific for it to respond to (since it relates to the remit of the sub-zone), it can simply point the querier to



Notes:

- ICANN manages assignment of and infrastructure for TLDs, and likewise for the root zone (i.e., the zone for the root of the name space); it does so as a sort of commercial stand-in for IANA.
- IANA maintain a definitive set of reserved TLDs [7] at

  http://www.iana.org/assignments/special-use-domain-names

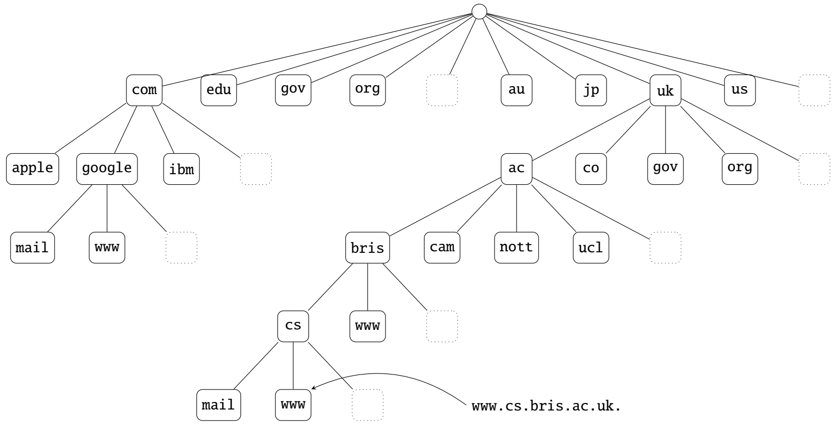
  which include examples such as arpa, which was originally used to transition the ARPANET to the Internet (i.e., all ARPANET names were initially "ported" into this part of the DNS name space).
- One can distinguish between a **Fully Qualified Domain Name (FQDN)** and **Partially Qualified Domain Name (PQDN)**: the former includes all components between root and host, whereas the latter does not (e.g., is relative to some base name). This concept is basically the same as in other (hierarchical) naming systems, e.g., a UNIX file system where absolute and relative paths are analogies.
- The majority of the uk TLD is managed by Nominet

  http://www.nominet.org.uk/

  with (some) exceptions including gov.uk and ac.uk, which is managed by JISC

  http://www.jisc.ac.uk/

- To (fully) specify the name of a leaf node, we simply traverse the name space tree toward the root adding a dot between the names at each level. The root node name is the empty string, meaning that *formally* the name produced will end with `.`; informally, we would typically omit this.
- DNS supports wildcard names, e.g., `*.cs.bris.ac.uk`; see [10] for a detailed overview.
- A DNS zone is simply a contiguous portion of the DNS name space, i.e., sub-tree of the hierarchy.
- Division of the name space into zones aims to offer scalability. The aim is to form a covering set of the entire name space: by having a queries relating to a given zone managed by a dedicated name server, this allows a) local management of the name space in that zone, b) provision of the infrastructure required to handle queries for that zone.
- In a sense, the approach of zoning in DNS is is somewhat similar to the use of subnetting with IP addresses: in both cases, the hierarchy imposed offers some help wrt. scalability. It's important to point out differences, however. For example, the DNS hierarchy could be described as logical in the sense there's no real reason `x.example.com` and `y.example.com` are connected in any sense than the name space; with an IP sub-net however, there must be a physical relationship of sorts because the approach to routing depends on it to some extent.
- In delegating from one zone to another (sub-)zone, the former basically acts as a pointer to the latter: if the zone name server is presented with a query too specific for it to respond to (since it relates to the remit of the sub-zone), it can simply point the querier to



Notes:

- ICANN manages assignment of and infrastructure for TLDs, and likewise for the root zone (i.e., the zone for the root of the name space); it does so as a sort of commercial stand-in for IANA.
- IANA maintain a definitive set of reserved TLDs [7] at

  http://www.iana.org/assignments/special-use-domain-names

  which include examples such as arpa, which was originally used to transition the ARPANET to the Internet (i.e., all ARPANET names were initially "ported" into this part of the DNS name space).
- One can distinguish between a **Fully Qualified Domain Name (FQDN)** and **Partially Qualified Domain Name (PQDN)**: the former includes all components between root and host, whereas the latter does not (e.g., is relative to some base name). This concept is basically the same as in other (hierarchical) naming systems, e.g., a UNIX file system where absolute and relative paths are analogies.
- The majority of the uk TLD is managed by Nominet

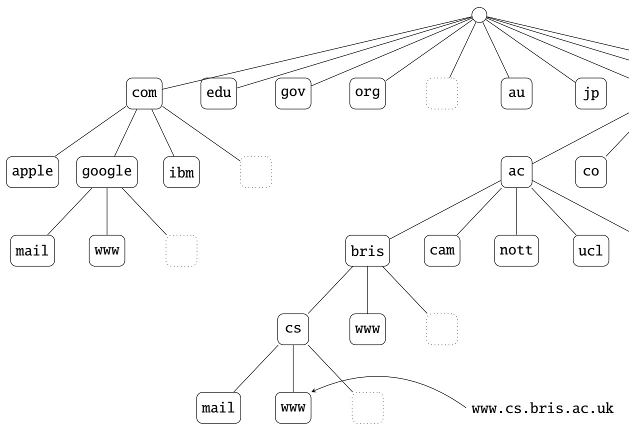
  http://www.nominet.org.uk/

  with (some) exceptions including gov.uk and ac.uk, which is managed by JISC

  http://www.jisc.ac.uk/

- To (fully) specify the name of a leaf node, we simply traverse the name space tree toward the root adding a dot between the names at each level. The root node name is the empty string, meaning that *formally* the name produced will end with `.`; informally, we would typically omit this.
- DNS supports wildcard names, e.g., `*.cs.bris.ac.uk`; see [10] for a detailed overview.
- A DNS zone is simply a contiguous portion of the DNS name space, i.e., sub-tree of the hierarchy.
- Division of the name space into zones aims to offer scalability. The aim is to form a covering set of the entire name space: by having a queries relating to a given zone managed by a dedicated name server, this allows a) local management of the name space in that zone, b) provision of the infrastructure required to handle queries for that zone.
- In a sense, the approach of zoning in DNS is is somewhat similar to the use of subnetting with IP addresses: in both cases, the hierarchy imposed offers some help wrt. scalability. It's important to point out differences, however. For example, the DNS hierarchy could be described as logical in the sense there's no real reason `x.example.com` and `y.example.com` are connected in any sense than the name space; with an IP sub-net however, there must be a physical relationship of sorts because the approach to routing depends on it to some extent.
- In delegating from one zone to another (sub-)zone, the former basically acts as a pointer to the latter: if the zone name server is presented with a query too specific for it to respond to (since it relates to the remit of the sub-zone), it can simply point the querier to

Problem #2 ⇝ DNS (3)



Notes:

- ICANN manages assignment of and infrastructure for TLDs, and likewise for the root zone (i.e., the zone for the root of the name space); it does so as a sort of commercial stand-in for IANA.
- IANA maintain a definitive set of reserved TLDs [7] at

  http://www.iana.org/assignments/special-use-domain-names

  which include examples such as arpa, which was originally used to transition the ARPANET to the Internet (i.e., all ARPANET names were initially "ported" into this part of the DNS name space).
- One can distinguish between a **Fully Qualified Domain Name (FQDN)** and **Partially Qualified Domain Name (PQDN)**: the former includes all components between root and host, whereas the latter does not (e.g., is relative to some base name). This concept is basically the same as in other (hierarchical) naming systems, e.g., a UNIX file system where absolute and relative paths are analogies.
- The majority of the uk TLD is managed by Nominet

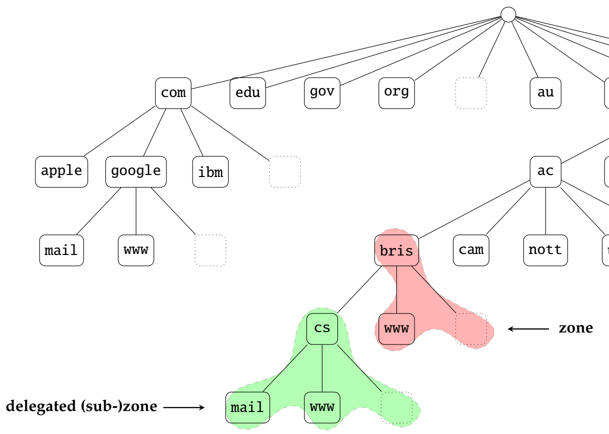
  http://www.nominet.org.uk/

  with (some) exceptions including gov.uk and ac.uk, which is managed by JISC

  http://www.jisc.ac.uk/

- To (fully) specify the name of a leaf node, we simply traverse the name space tree toward the root adding a dot between the names at each level. The root node name is the empty string, meaning that *formally* the name produced will end with .; informally, we would typically omit this.
- DNS supports wildcard names, e.g., *.cs.bris.ac.uk; see [10] for a detailed overview.
- A DNS zone is simply a contiguous portion of the DNS name space, i.e., sub-tree of the hierarchy.
- Division of the name space into zones aims to offer scalability. The aim is to form a covering set of the entire name space: by having a queries relating to a given zone managed by a dedicated name server, this allows a) local management of the name space in that zone, b) provision of the infrastructure required to handle queries for that zone.
- In a sense, the approach of zoning in DNS is is somewhat similar to the use of subnetting with IP addresses: in both cases, the hierarchy imposed offers some help wrt. scalability. It's important to point out differences, however. For example, the DNS hierarchy could be described as logical in the sense there's no real reason x.example.com and y.example.com are connected in any sense than the name space; with an IP sub-net however, there must be a physical relationship of sorts because the approach to routing depends on it to some extent.
- In delegating from one zone to another (sub-)zone, the former basically acts as a pointer to the latter: if the zone name server is presented with a query too specific for it to respond to (since it relates to the remit of the sub-zone), it can simply point the querier to

Problem #2 ⇝ DNS (3)



Notes:

- ICANN manages assignment of and infrastructure for TLDs, and likewise for the root zone (i.e., the zone for the root of the name space); it does so as a sort of commercial stand-in for IANA.
- IANA maintain a definitive set of reserved TLDs [7] at

  http://www.iana.org/assignments/special-use-domain-names

  which include examples such as arpa, which was originally used to transition the ARPANET to the Internet (i.e., all ARPANET names were initially "ported" into this part of the DNS name space).
- One can distinguish between a **Fully Qualified Domain Name (FQDN)** and **Partially Qualified Domain Name (PQDN)**: the former includes all components between root and host, whereas the latter does not (e.g., is relative to some base name). This concept is basically the same as in other (hierarchical) naming systems, e.g., a UNIX file system where absolute and relative paths are analogies.
- The majority of the uk TLD is managed by Nominet

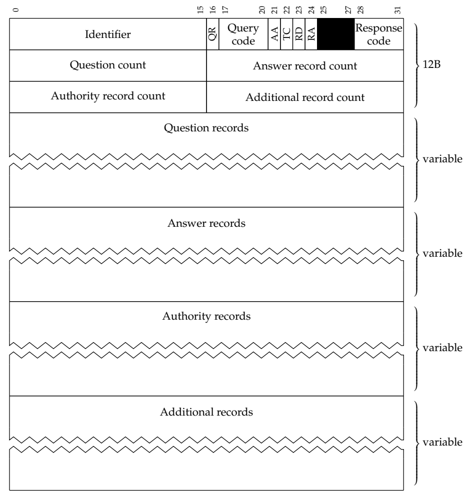
  http://www.nominet.org.uk/

  with (some) exceptions including gov.uk and ac.uk, which is managed by JISC

  http://www.jisc.ac.uk/

- To (fully) specify the name of a leaf node, we simply traverse the name space tree toward the root adding a dot between the names at each level. The root node name is the empty string, meaning that *formally* the name produced will end with .; informally, we would typically omit this.
- DNS supports wildcard names, e.g., *.cs.bris.ac.uk; see [10] for a detailed overview.
- A DNS zone is simply a contiguous portion of the DNS name space, i.e., sub-tree of the hierarchy.
- Division of the name space into zones aims to offer scalability. The aim is to form a covering set of the entire name space: by having a queries relating to a given zone managed by a dedicated name server, this allows a) local management of the name space in that zone, b) provision of the infrastructure required to handle queries for that zone.
- In a sense, the approach of zoning in DNS is is somewhat similar to the use of subnetting with IP addresses: in both cases, the hierarchy imposed offers some help wrt. scalability. It's important to point out differences, however. For example, the DNS hierarchy could be described as logical in the sense there's no real reason x.example.com and y.example.com are connected in any sense than the name space; with an IP sub-net however, there must be a physical relationship of sorts because the approach to routing depends on it to some extent.
- In delegating from one zone to another (sub-)zone, the former basically acts as a pointer to the latter: if the zone name server is presented with a query too specific for it to respond to (since it relates to the remit of the sub-zone), it can simply point the querier to

Data Structure (**DNS message** [12])



The data structure includes:

- A 16-bit identifier used to link query and response messages.
- 4-bit query and response codes.
- A set of flags, including
  - 1-bit **Query/Response (QR)** flag, to disambiguate queries from responses,
  - 1-bit **Authoritative Answer (AA)** flag, which marks responses that are authoritative,
  - 1-bit **Truncation Flag (TC)** which marks responses longer than the UDP 512B limit,
  - 1-bit **Recursion Desired (RD)** flag, used to request recursive resolution,
  - 1-bit **Recursion Available (RA)** flag, used to signal availability of recursive resolution.
- Some number of records (of each type).

Notes:

- The message (and record) data structure(s) are, of course, for communication; the DNS server itself obviously needs to maintain a database of records that can be queried in order to construct suitable responses. [12] includes a text-based **master file** format for this purpose.
- IANA maintain a definitive set of assigned DNS parameters, e.g., query and response codes, at

  `http://www.iana.org/assignments/dns-parameters`

- The encoding of names within various query and response types might differ from what you'd expect. Ignoring issues such as character encoding (which is ASCII), DNS basically just needs to encode a string; any valid encoding would be fine, so we could follow C and adopt a null-terminated. We would have, for example

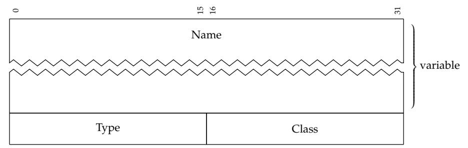
  `www.cs.bris.ac.uk` $\mapsto$ ⟨'w', 'w', 'w', '.', 'c', 's', '.', 'b', 'r', 'i', 's', '.', 'a', 'c', '.', 'u', 'k', '.', 0⟩

  However, parsing this is problematic: we need to find '.' separators. So instead, DNS uses a length-prefixed alternative st.

  `www.cs.bris.ac.uk` $\mapsto$ ⟨3, 'w', 'w', 'w', 2, 'c', 's', 4, 'b', 'r', 'i', 's', 2, 'a', 'c', 2, 'u', 'k', 0⟩

  which is much easier to parse step-by-step. Where an email address forms part of a record, the same encoding is used; the '' which separates the user name from domain name is replaced with a '.'.
  The encoding *also* supports a form of dictionary-based compression for names, which aims to reduce the message length when multiple, similar names are included in a record. The idea is to allow an embedded pointer in one encoded name, which links to another.
- As you might expect, a zero TTL suggests the record should not be cached.

Data Structure (**DNS question record** [12])



The data structure includes:

- A variable length, length-prefixed name string.
- A 16-bit record type.
- A 16-bit record class.

Notes:

- The message (and record) data structure(s) are, of course, for communication; the DNS server itself obviously needs to maintain a database of records that can be queried in order to construct suitable responses. [12] includes a text-based **master file** format for this purpose.
- IANA maintain a definitive set of assigned DNS parameters, e.g., query and response codes, at

  `http://www.iana.org/assignments/dns-parameters`

- The encoding of names within various query and response types might differ from what you'd expect. Ignoring issues such as character encoding (which is ASCII), DNS basically just needs to encode a string; any valid encoding would be fine, so we could follow C and adopt a null-terminated. We would have, for example

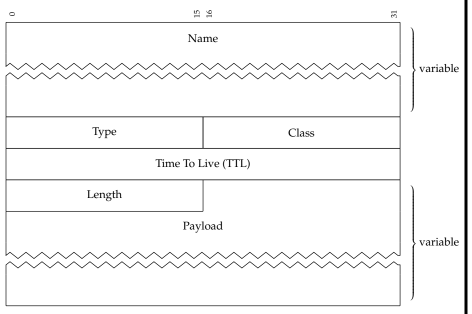
  `www.cs.bris.ac.uk` $\mapsto$ ⟨'w', 'w', 'w', '.', 'c', 's', '.', 'b', 'r', 'i', 's', '.', 'a', 'c', '.', 'u', 'k', '.', 0⟩

  However, parsing this is problematic: we need to find '.' separators. So instead, DNS uses a length-prefixed alternative st.

  `www.cs.bris.ac.uk` $\mapsto$ ⟨3, 'w', 'w', 'w', 2, 'c', 's', 4, 'b', 'r', 'i', 's', 2, 'a', 'c', 2, 'u', 'k', 0⟩

  which is much easier to parse step-by-step. Where an email address forms part of a record, the same encoding is used; the '' which separates the user name from domain name is replaced with a '.'.
  The encoding *also* supports a form of dictionary-based compression for names, which aims to reduce the message length when multiple, similar names are included in a record. The idea is to allow an embedded pointer in one encoded name, which links to another.
- As you might expect, a zero TTL suggests the record should not be cached.

Data Structure (DNS resource record [12])



The data structure includes:

- A variable length, length-prefixed name string.
- A 16-bit record type.
- A 16-bit record class.
- A 32-bit **Time To Live (TTL)** field which controls cache retention.
- A 16-bit record length, and the record-specific payload, e.g.,
  1. a 32-bit IP address for A records,
  2. a variable length name for NS records, and
  3. a variable length name for CNAME records.

Notes:

- The message (and record) data structure(s) are, of course, for communication; the DNS server itself obviously needs to maintain a database of records that can be queried in order to construct suitable responses. [12] includes a text-based **master file** format for this purpose.
- IANA maintain a definitive set of assigned DNS parameters, e.g., query and response codes, at

  `http://www.iana.org/assignments/dns-parameters`

- The encoding of names within various query and response types might differ from what you'd expect. Ignoring issues such as character encoding (which is ASCII), DNS basically just needs to encode a string; any valid encoding would be fine, so we could follow C and adopt a null-terminated. We would have, for example

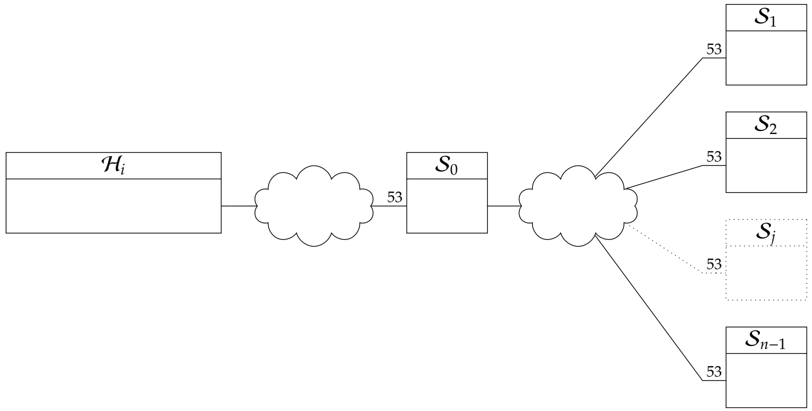
  `www.cs.bris.ac.uk` ↦ ⟨'w', 'w', 'w', '.', 'c', 's', '.', 'b', 'r', 'i', 's', '.', 'a', 'c', '.', 'u', 'k', '.', 0⟩

  However, parsing this is problematic: we need to find '.' separators. So instead, DNS uses a length-prefixed alternative st.

  `www.cs.bris.ac.uk` ↦ ⟨3, 'w', 'w', 'w', 2, 'c', 's', 4, 'b', 'r', 'i', 's', 2, 'a', 'c', 2, 'u', 'k', 0⟩

  which is much easier to parse step-by-step. Where an email address forms part of a record, the same encoding is used; the '' which separates the user name from domain name is replaced with a '.'.
  The encoding *also* supports a form of dictionary-based compression for names, which aims to reduce the message length when multiple, similar names are included in a record. The idea is to allow an embedded pointer in one encoded name, which links to another.
- As you might expect, a zero TTL suggests the record should not be cached.

- Example:



Notes:

- 13 root name servers offer authoritative responses to queries wrt. the TLDs; the servers are named `a.root-servers.net` to `m.root-servers.net`, and are replicated geographically to support load balancing (via use of anycast). The web-site

  `http://root-servers.org/`

  provides an easy way to visualise where (geographically) the root name servers are and who operates them.
- For this mechanism to operate, one needs to bootstrap
  - the host, so it knows the IP address of the local name server (this is often provided when an IP address is allocated via DHCP),
  - the local name server, so it knows the IP addresses of at least one root name server, and
  - the authoritative name server, so it knows the definitive response for a given query
- Notice that
  - a recursive query basically asks the name server to do *all* the work required to resolve a name, while
  - a recursive query basically asks the name server to do *just* one step required to resolve a name.

  But division between recursive and iterative queries might *seem* confusing; the (or one) reason to separate them is so a name server can be optimised wrt. one or other type, e.g.,
  - supporting recursive queries allows the name server to support **caching** (while respecting a TTL that allows cached responses to expire),
  - supporting iterative queries allows the name server to avoid any state wrt. the connection (the query/response is one-shot), and perform load balancing more easily.

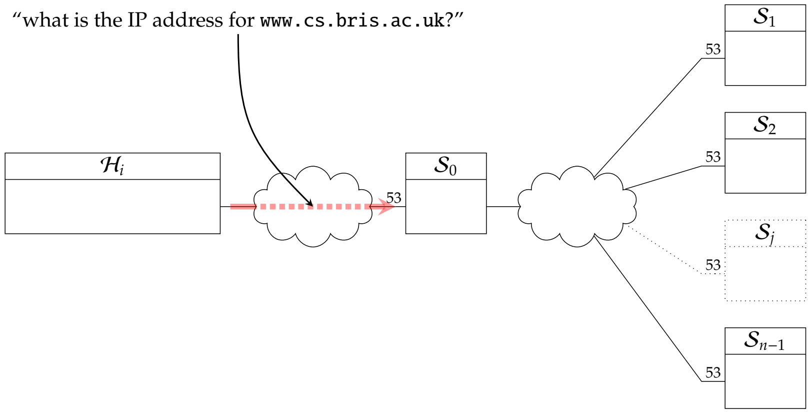
  Therefore, local name servers typically allow recursive queries (since this makes them more efficient, and useful for hosts) while those that support higher levels of the name space allow iterative queries alone.
- The figure shows the local name server as a separate host, queried by a so-called stub resolver on the host: in Linux, configuration of the C standard library resolver is exposed via

  `/etc/resolv.conf`

  for example. In reality, however, there is no reason why a recursive name server cannot be executed on the host itself, and [12] illustrates various organisations.
  Even then "local" doesn't *necessarily* imply the server used is geographically close; really all the host needs is *a* name server willing to respond to recursive queries. The Google Public DNS service operates such a server at `8.8.8.8`, meaning one could use this as the local name server, for example.

Problem #2 $\rightsquigarrow$ DNS (5)

- **Example**:



Notes:

- 13 root name servers offer authoritative responses to queries wrt. the TLDs; the servers are named `a.root-servers.net` to `m.root-servers.net`, and are replicated geographically to support load balancing (via use of anycast). The web-site

  http://root-servers.org/

  provides an easy way to visualise where (geographically) the root name servers are and who operates them.
- For this mechanism to operate, one needs to bootstrap
  - the host, so it knows the IP address of the local name server (this is often provided when an IP address is allocated via DHCP),
  - the local name server, so it knows the IP addresses of at least one root name server, and
  - the authoritative name server, so it knows the definitive response for a given query
- Notice that
  - a recursive query basically asks the name server to do *all* the work required to resolve a name, while
  - a recursive query basically asks the name server to do *just* one step required to resolve a name.

  But division between recursive and iterative queries might *seem* confusing; the (or one) reason to separate them is so a name server can be optimised wrt. one or other type, e.g.,
  - supporting recursive queries allows the name server to support **caching** (while respecting a TTL that allows cached responses to expire),
  - supporting iterative queries allows the name server to avoid any state wrt. the connection (the query/response is one-shot), and perform load balancing more easily.

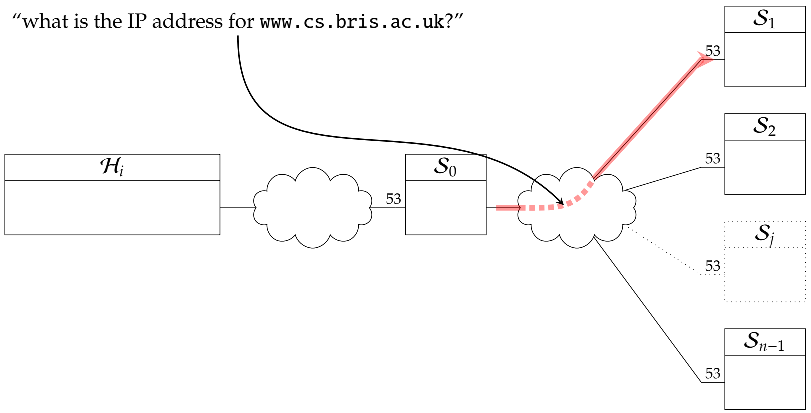
  Therefore, local name servers typically allow recursive queries (since this makes them more efficient, and useful for hosts) while those that support higher levels of the name space allow iterative queries alone.
- The figure shows the local name server as a separate host, queried by a so-called stub resolver on the host: in Linux, configuration of the C standard library resolver is exposed via

  `/etc/resolv.conf`

  for example. In reality, however, there is no reason why a recursive name server cannot be executed on the host itself, and [12] illustrates various organisations.
  Even then "local" doesn't *necessarily* imply the server used is geographically close; really all the host needs is *a* name server willing to respond to recursive queries. The Google Public DNS service operates such a server at `8.8.8.8`, meaning one could use this as the local name server, for example.

Problem #2 $\rightsquigarrow$ DNS (5)

- **Example**:



Notes:

- 13 root name servers offer authoritative responses to queries wrt. the TLDs; the servers are named `a.root-servers.net` to `m.root-servers.net`, and are replicated geographically to support load balancing (via use of anycast). The web-site

  http://root-servers.org/

  provides an easy way to visualise where (geographically) the root name servers are and who operates them.
- For this mechanism to operate, one needs to bootstrap
  - the host, so it knows the IP address of the local name server (this is often provided when an IP address is allocated via DHCP),
  - the local name server, so it knows the IP addresses of at least one root name server, and
  - the authoritative name server, so it knows the definitive response for a given query
- Notice that
  - a recursive query basically asks the name server to do *all* the work required to resolve a name, while
  - a recursive query basically asks the name server to do *just* one step required to resolve a name.

  But division between recursive and iterative queries might *seem* confusing; the (or one) reason to separate them is so a name server can be optimised wrt. one or other type, e.g.,
  - supporting recursive queries allows the name server to support **caching** (while respecting a TTL that allows cached responses to expire),
  - supporting iterative queries allows the name server to avoid any state wrt. the connection (the query/response is one-shot), and perform load balancing more easily.

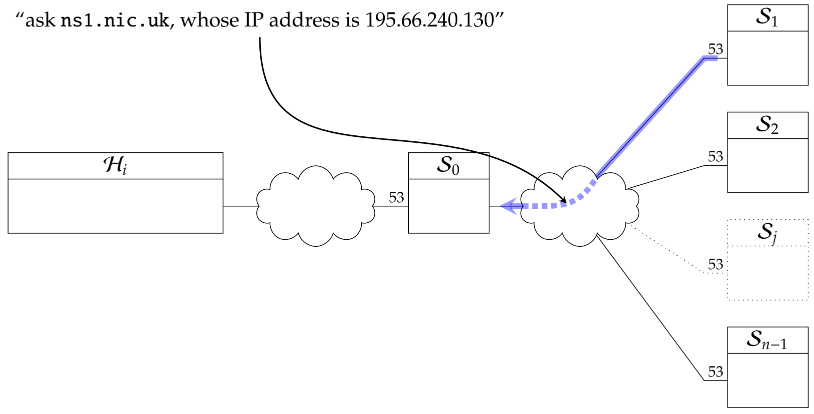
  Therefore, local name servers typically allow recursive queries (since this makes them more efficient, and useful for hosts) while those that support higher levels of the name space allow iterative queries alone.
- The figure shows the local name server as a separate host, queried by a so-called stub resolver on the host: in Linux, configuration of the C standard library resolver is exposed via

  `/etc/resolv.conf`

  for example. In reality, however, there is no reason why a recursive name server cannot be executed on the host itself, and [12] illustrates various organisations.
  Even then "local" doesn't *necessarily* imply the server used is geographically close; really all the host needs is *a* name server willing to respond to recursive queries. The Google Public DNS service operates such a server at `8.8.8.8`, meaning one could use this as the local name server, for example.

Problem #2 ⇝ DNS (5)

- Example:



Notes:

- 13 root name servers offer authoritative responses to queries wrt. the TLDs; the servers are named `a.root-servers.net` to `m.root-servers.net`, and are replicated geographically to support load balancing (via use of anycast). The web-site

  `http://root-servers.org/`

  provides an easy way to visualise where (geographically) the root name servers are and who operates them.
- For this mechanism to operate, one needs to bootstrap
  - the host, so it knows the IP address of the local name server (this is often provided when an IP address is allocated via DHCP),
  - the local name server, so it knows the IP addresses of at least one root name server, and
  - the authoritative name server, so it knows the definitive response for a given query
- Notice that
  - a recursive query basically asks the name server to do *all* the work required to resolve a name, while
  - a recursive query basically asks the name server to do *just* one step required to resolve a name.

  But division between recursive and iterative queries might *seem* confusing; the (or one) reason to separate them is so a name server can be optimised wrt. one or other type, e.g.,
  - supporting recursive queries allows the name server to support **caching** (while respecting a TTL that allows cached responses to expire),
  - supporting iterative queries allows the name server to avoid any state wrt. the connection (the query/response is one-shot), and perform load balancing more easily.

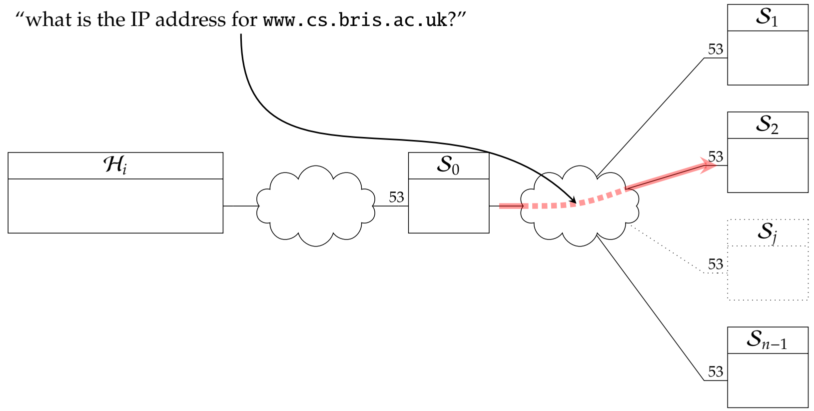
  Therefore, local name servers typically allow recursive queries (since this makes them more efficient, and useful for hosts) while those that support higher levels of the name space allow iterative queries alone.
- The figure shows the local name server as a separate host, queried by a so-called stub resolver on the host: in Linux, configuration of the C standard library resolver is exposed via

  `/etc/resolv.conf`

  for example. In reality, however, there is no reason why a recursive name server cannot be executed on the host itself, and [12] illustrates various organisations.
  Even then "local" doesn't *necessarily* imply the server used is geographically close; really all the host needs is *a* name server willing to respond to recursive queries. The Google Public DNS service operates such a server at `8.8.8.8`, meaning one could use this as the local name server, for example.

Problem #2 ⇝ DNS (5)

- Example:



Notes:

- 13 root name servers offer authoritative responses to queries wrt. the TLDs; the servers are named `a.root-servers.net` to `m.root-servers.net`, and are replicated geographically to support load balancing (via use of anycast). The web-site

  `http://root-servers.org/`

  provides an easy way to visualise where (geographically) the root name servers are and who operates them.
- For this mechanism to operate, one needs to bootstrap
  - the host, so it knows the IP address of the local name server (this is often provided when an IP address is allocated via DHCP),
  - the local name server, so it knows the IP addresses of at least one root name server, and
  - the authoritative name server, so it knows the definitive response for a given query
- Notice that
  - a recursive query basically asks the name server to do *all* the work required to resolve a name, while
  - a recursive query basically asks the name server to do *just* one step required to resolve a name.

  But division between recursive and iterative queries might *seem* confusing; the (or one) reason to separate them is so a name server can be optimised wrt. one or other type, e.g.,
  - supporting recursive queries allows the name server to support **caching** (while respecting a TTL that allows cached responses to expire),
  - supporting iterative queries allows the name server to avoid any state wrt. the connection (the query/response is one-shot), and perform load balancing more easily.

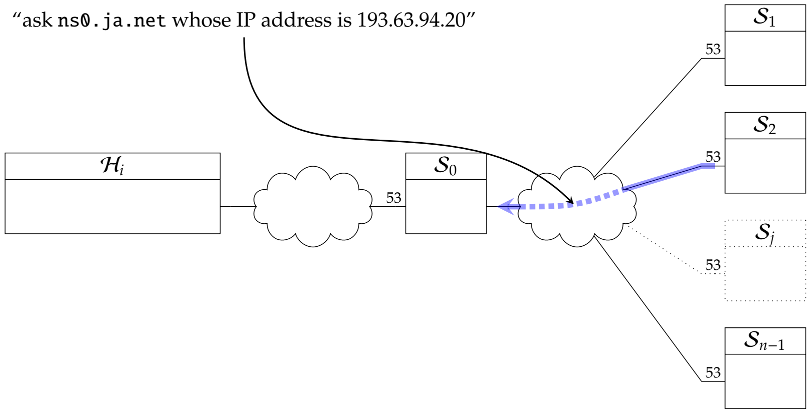
  Therefore, local name servers typically allow recursive queries (since this makes them more efficient, and useful for hosts) while those that support higher levels of the name space allow iterative queries alone.
- The figure shows the local name server as a separate host, queried by a so-called stub resolver on the host: in Linux, configuration of the C standard library resolver is exposed via

  `/etc/resolv.conf`

  for example. In reality, however, there is no reason why a recursive name server cannot be executed on the host itself, and [12] illustrates various organisations.
  Even then "local" doesn't *necessarily* imply the server used is geographically close; really all the host needs is *a* name server willing to respond to recursive queries. The Google Public DNS service operates such a server at `8.8.8.8`, meaning one could use this as the local name server, for example.

- Example:

"ask `ns0.ja.net` whose IP address is 193.63.94.20"



Notes:

- 13 root name servers offer authoritative responses to queries wrt. the TLDs; the servers are named `a.root-servers.net` to `m.root-servers.net`, and are replicated geographically to support load balancing (via use of anycast). The web-site

  `http://root-servers.org/`

  provides an easy way to visualise where (geographically) the root name servers are and who operates them.
- For this mechanism to operate, one needs to bootstrap
  - the host, so it knows the IP address of the local name server (this is often provided when an IP address is allocated via DHCP),
  - the local name server, so it knows the IP addresses of at least one root name server, and
  - the authoritative name server, so it knows the definitive response for a given query
- Notice that
  - a recursive query basically asks the name server to do *all* the work required to resolve a name, while
  - a recursive query basically asks the name server to do *just* one step required to resolve a name.

  But division between recursive and iterative queries might *seem* confusing; the (or one) reason to separate them is so a name server can be optimised wrt. one or other type, e.g.,
  - supporting recursive queries allows the name server to support **caching** (while respecting a TTL that allows cached responses to expire),
  - supporting iterative queries allows the name server to avoid any state wrt. the connection (the query/response is one-shot), and perform load balancing more easily.

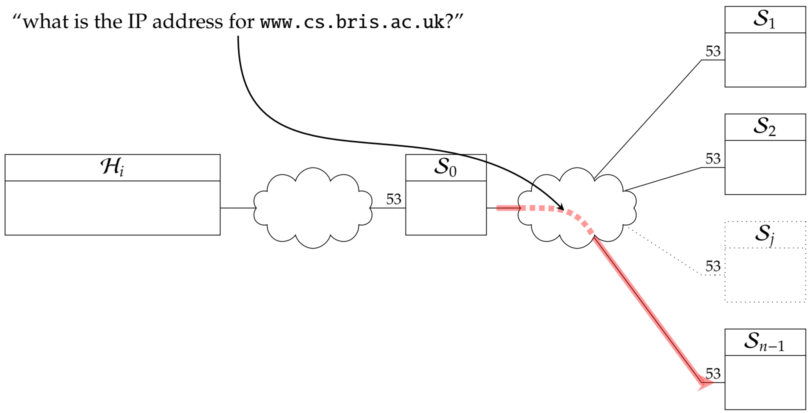
  Therefore, local name servers typically allow recursive queries (since this makes them more efficient, and useful for hosts) while those that support higher levels of the name space allow iterative queries alone.
- The figure shows the local name server as a separate host, queried by a so-called stub resolver on the host: in Linux, configuration of the C standard library resolver is exposed via

  `/etc/resolv.conf`

  for example. In reality, however, there is no reason why a recursive name server cannot be executed on the host itself, and [12] illustrates various organisations.
  Even then "local" doesn't *necessarily* imply the server used is geographically close; really all the host needs is *a* name server willing to respond to recursive queries. The Google Public DNS service operates such a server at `8.8.8.8`, meaning one could use this as the local name server, for example.

- Example:

"what is the IP address for `www.cs.bris.ac.uk`?"



Notes:

- 13 root name servers offer authoritative responses to queries wrt. the TLDs; the servers are named `a.root-servers.net` to `m.root-servers.net`, and are replicated geographically to support load balancing (via use of anycast). The web-site

  `http://root-servers.org/`

  provides an easy way to visualise where (geographically) the root name servers are and who operates them.
- For this mechanism to operate, one needs to bootstrap
  - the host, so it knows the IP address of the local name server (this is often provided when an IP address is allocated via DHCP),
  - the local name server, so it knows the IP addresses of at least one root name server, and
  - the authoritative name server, so it knows the definitive response for a given query
- Notice that
  - a recursive query basically asks the name server to do *all* the work required to resolve a name, while
  - a recursive query basically asks the name server to do *just* one step required to resolve a name.

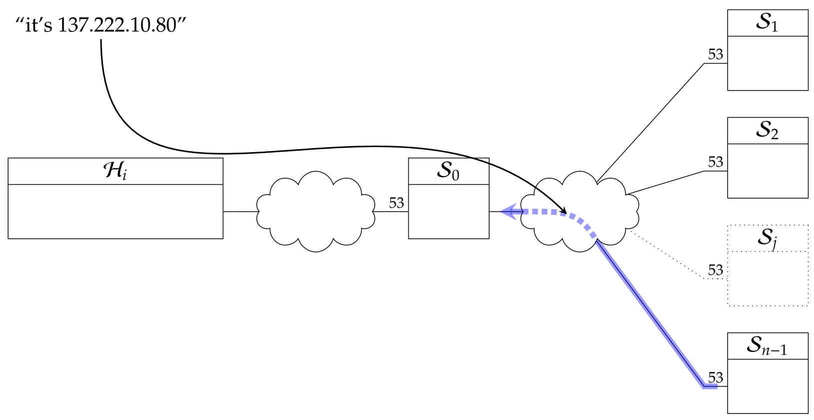
  But division between recursive and iterative queries might *seem* confusing; the (or one) reason to separate them is so a name server can be optimised wrt. one or other type, e.g.,
  - supporting recursive queries allows the name server to support **caching** (while respecting a TTL that allows cached responses to expire),
  - supporting iterative queries allows the name server to avoid any state wrt. the connection (the query/response is one-shot), and perform load balancing more easily.

  Therefore, local name servers typically allow recursive queries (since this makes them more efficient, and useful for hosts) while those that support higher levels of the name space allow iterative queries alone.
- The figure shows the local name server as a separate host, queried by a so-called stub resolver on the host: in Linux, configuration of the C standard library resolver is exposed via

  `/etc/resolv.conf`

  for example. In reality, however, there is no reason why a recursive name server cannot be executed on the host itself, and [12] illustrates various organisations.
  Even then "local" doesn't *necessarily* imply the server used is geographically close; really all the host needs is *a* name server willing to respond to recursive queries. The Google Public DNS service operates such a server at `8.8.8.8`, meaning one could use this as the local name server, for example.

- Example:



Notes:

- 13 root name servers offer authoritative responses to queries wrt. the TLDs; the servers are named `a.root-servers.net` to `m.root-servers.net`, and are replicated geographically to support load balancing (via use of anycast). The web-site

  http://root-servers.org/

  provides an easy way to visualise where (geographically) the root name servers are and who operates them.
- For this mechanism to operate, one needs to bootstrap
  - the host, so it knows the IP address of the local name server (this is often provided when an IP address is allocated via DHCP),
  - the local name server, so it knows the IP addresses of at least one root name server, and
  - the authoritative name server, so it knows the definitive response for a given query
- Notice that
  - a recursive query basically asks the name server to do *all* the work required to resolve a name, while
  - a recursive query basically asks the name server to do *just* one step required to resolve a name.

  But division between recursive and iterative queries might *seem* confusing; the (or one) reason to separate them is so a name server can be optimised wrt. one or other type, e.g.,
  - supporting recursive queries allows the name server to support **caching** (while respecting a TTL that allows cached responses to expire),
  - supporting iterative queries allows the name server to avoid any state wrt. the connection (the query/response is one-shot), and perform load balancing more easily.

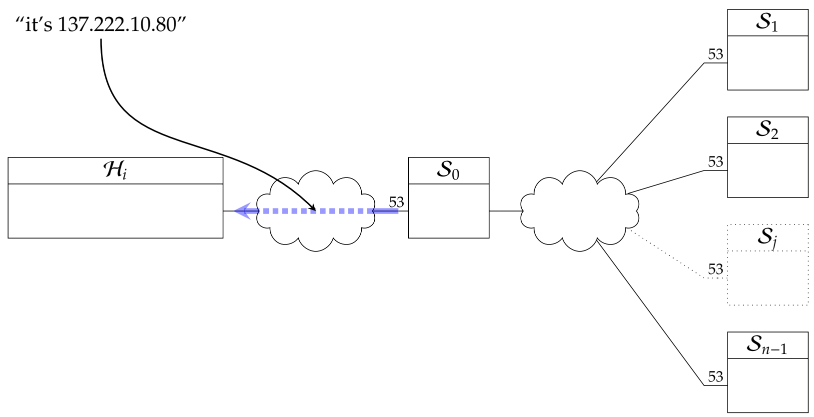
  Therefore, local name servers typically allow recursive queries (since this makes them more efficient, and useful for hosts) while those that support higher levels of the name space allow iterative queries alone.
- The figure shows the local name server as a separate host, queried by a so-called stub resolver on the host: in Linux, configuration of the C standard library resolver is exposed via

  `/etc/resolv.conf`

  for example. In reality, however, there is no reason why a recursive name server cannot be executed on the host itself, and [12] illustrates various organisations.
  Even then "local" doesn't *necessarily* imply the server used is geographically close; really all the host needs is *a* name server willing to respond to recursive queries. The Google Public DNS service operates such a server at `8.8.8.8`, meaning one could use this as the local name server, for example.

Problem #2 ⇝ DNS (5)

- Example:



Notes:

- 13 root name servers offer authoritative responses to queries wrt. the TLDs; the servers are named `a.root-servers.net` to `m.root-servers.net`, and are replicated geographically to support load balancing (via use of anycast). The web-site

  http://root-servers.org/

  provides an easy way to visualise where (geographically) the root name servers are and who operates them.
- For this mechanism to operate, one needs to bootstrap
  - the host, so it knows the IP address of the local name server (this is often provided when an IP address is allocated via DHCP),
  - the local name server, so it knows the IP addresses of at least one root name server, and
  - the authoritative name server, so it knows the definitive response for a given query
- Notice that
  - a recursive query basically asks the name server to do *all* the work required to resolve a name, while
  - a recursive query basically asks the name server to do *just* one step required to resolve a name.

  But division between recursive and iterative queries might *seem* confusing; the (or one) reason to separate them is so a name server can be optimised wrt. one or other type, e.g.,
  - supporting recursive queries allows the name server to support **caching** (while respecting a TTL that allows cached responses to expire),
  - supporting iterative queries allows the name server to avoid any state wrt. the connection (the query/response is one-shot), and perform load balancing more easily.

  Therefore, local name servers typically allow recursive queries (since this makes them more efficient, and useful for hosts) while those that support higher levels of the name space allow iterative queries alone.
- The figure shows the local name server as a separate host, queried by a so-called stub resolver on the host: in Linux, configuration of the C standard library resolver is exposed via

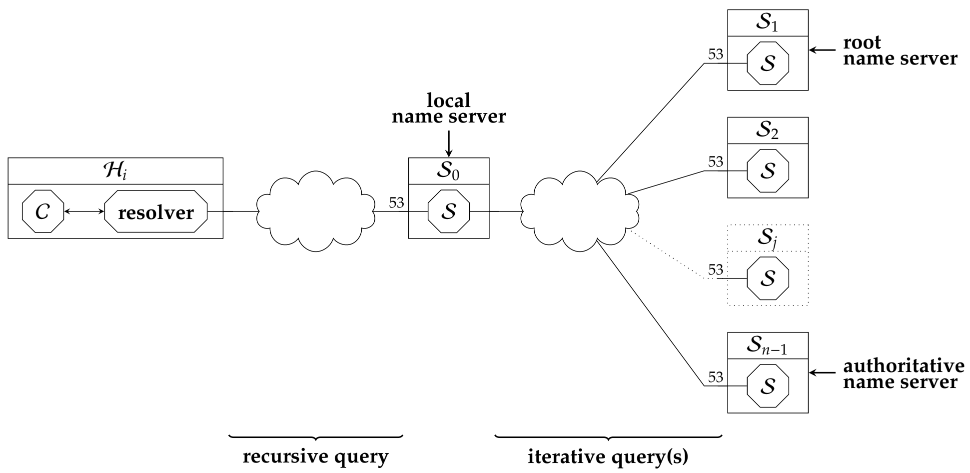
  `/etc/resolv.conf`

  for example. In reality, however, there is no reason why a recursive name server cannot be executed on the host itself, and [12] illustrates various organisations.
  Even then "local" doesn't *necessarily* imply the server used is geographically close; really all the host needs is *a* name server willing to respond to recursive queries. The Google Public DNS service operates such a server at `8.8.8.8`, meaning one could use this as the local name server, for example.

Daniel Page (Daniel.Page@bristol.ac.uk)
Concurrent Computing (Computer Networks) git # 3627080 @ 2016-03-11
University of BRISTOL

Problem #2 ⇝ DNS (5)

- Example:



Notes:

- 13 root name servers offer authoritative responses to queries wrt. the TLDs; the servers are named `a.root-servers.net` to `m.root-servers.net`, and are replicated geographically to support load balancing (via use of anycast). The web-site

  `http://root-servers.org/`

  provides an easy way to visualise where (geographically) the root name servers are and who operates them.
- For this mechanism to operate, one needs to bootstrap
  - the host, so it knows the IP address of the local name server (this is often provided when an IP address is allocated via DHCP),
  - the local name server, so it knows the IP addresses of at least one root name server, and
  - the authoritative name server, so it knows the definitive response for a given query
- Notice that
  - a recursive query basically asks the name server to do *all* the work required to resolve a name, while
  - a recursive query basically asks the name server to do *just* one step required to resolve a name.

  But division between recursive and iterative queries might *seem* confusing; the (or one) reason to separate them is so a name server can be optimised wrt. one or other type, e.g.,
  - supporting recursive queries allows the name server to support **caching** (while respecting a TTL that allows cached responses to expire),
  - supporting iterative queries allows the name server to avoid any state wrt. the connection (the query/response is one-shot), and perform load balancing more easily.

  Therefore, local name servers typically allow recursive queries (since this makes them more efficient, and useful for hosts) while those that support higher levels of the name space allow iterative queries alone.
- The figure shows the local name server as a separate host, queried by a so-called stub resolver on the host: in Linux, configuration of the C standard library resolver is exposed via

  `/etc/resolv.conf`

  for example. In reality, however, there is no reason why a recursive name server cannot be executed on the host itself, and [12] illustrates various organisations.
  Even then "local" doesn't *necessarily* imply the server used is geographically close; really all the host needs is *a* name server willing to respond to recursive queries. The Google Public DNS service operates such a server at `8.8.8.8`, meaning one could use this as the local name server, for example.

Conclusions

- Take away points:
  - The "glue" protocols outlined here solve issues that stem from practical deployment and use.
  - DNS in particular is interesting, since
    - it highlights the tricky compromise enforced by lack of middle-ground between UDP and TCP,
    - like DHCP *users* consider it part of the network but in reality it is an application level mechanism,
    - although not *too* old, it already highlights the need for flexibility to meet changing requirements and use-cases,
    - versus other glue protocols, there is a much stronger efficiency requirement, plus
    - tangential requirements (e.g., security [5], cf. DNSSEC [4] etc.) make it a *much* larger and more challenging problem.

Notes:

References

[1] Wikipedia: Domain name.
http://en.wikipedia.org/wiki/Domain_name.

[2] Wikipedia: Domain Name System (DNS).
http://en.wikipedia.org/wiki/Domain_Name_System.

[3] Wikipedia: Network address translation.
http://en.wikipedia.org/wiki/Network_address_translation.

[4] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose.
DNS security introduction and requirements.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 4033, 2005.
http://tools.ietf.org/html/rfc4033.

[5] D. Atkins and R. Austein.
Threat analysis of the Domain Name System (DNS).
Internet Engineering Task Force (IETF) Request for Comments (RFC) 3833, 2004.
http://tools.ietf.org/html/rfc3833.

[6] B. Carpenter and S. Brim.
Middleboxes: Taxonomy and issues.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 3234, 2002.
http://tools.ietf.org/html/rfc3234.

Notes:

References

[7] D. Eastlake and A. Panitz.
Reserved top level DNS names.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 2606, 1999.
http://tools.ietf.org/html/rfc2606.

[8] K. Egevang and P. Francis.
The IP Network Address Translator (NAT).
Internet Engineering Task Force (IETF) Request for Comments (RFC) 1631, 1994.
http://tools.ietf.org/html/rfc1631.

[9] E. Feinler, K. Harrenstien, Z.-S. Su, and V. White.
DoD Internet host table specification.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 810, 1982.
http://tools.ietf.org/html/rfc810.

[10] E. Lewis.
The role of wildcards in the domain name system.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 4592, 2006.
http://tools.ietf.org/html/rfc4592.

[11] P. Mockapetris.
Domain names - concepts and facilities.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 882, 1983.
http://tools.ietf.org/html/rfc882.

Notes:

[12] P. Mockapetris.
Domain names - implementation and specification.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 883, 1983.
http://tools.ietf.org/html/rfc883.

Notes: