INS: Internet Systems

Contents

Brief Introduction to Everything

Architecture

Networks consist of **interconnected nodes** that fulfil various roles: simple hosts, routers (which communicate with and send messages between hosts), servers (which provide some service), etc.

A **connection** does not imply **communication**: a **protocol** is needed for that. A protocol is an agreed schema for sending messages between nodes.

A **Local Area Network (LAN)** connects machines within some finite localisation, usually a physical location (e.g. an office building).

A **Wide Area Network (WAN)** connects various LANs together over larger areas, usually not directly (i.e. a small number of connections between LANs).

Routing is required because data may have to travel between various intermediate nodes. These nodes need to know that the data is not for them, and where they should send it. **Internet Protocol (IP) addresses** are used for this.

History of the Internet

The very first point-to-point connections were created by the **Advanced Research Projects Agency (ARPA)** in the 1960s. ARPA found some problems with early networking models:

- Centralisation created bottle-necks and high-risk failure points.
- Different LANs used various different protocols, creating compatibility issues.
- Commercial protocols were expensive to use.

They created **ARPANET** to solve these problems. ARPANET and the current Internet are fundamentally unreliable:

- Unreliable delivery means that only a best effort is made to deliver packets (i.e. no guarantee is made).
- Packets can be dropped with no notification to sender or receiver.
- Software must be able to deal with lost data.

Addressing

Resource addressing on the Internet is **hierarchical**: addresses are formed of **nested layers**, from port numbers on a machine and its physical wiring, up to access layers, inter-network layers and application layers.

Addresses only need to be unique **within their layer** and can be duplicated in other layers. The whole 'chain' of a given address is what needs to be unique.

See more: Addressing, page 13

Messaging Protocols

Messaging protocols are needed so that machines can be programmed to **understand each other**. Messages in one protocol can be **embedded** into another, such as:

$$Header_{protocol1} \cdot Message_{protocol2} \cdot Tail_{protocol1}$$

Messages will usually be **chunked**, so **flow control** is needed to deal with bandwidth, scheduling, routing, etc. The buffer size of the receiver must be considered. The receiver may drop packets that are sent too fast and ask the sender to slow down – how this happens will depend on the protocol, but **Transmission Control Protocol (TCP)** has this capability.

See more: Transmission Control Protocol (TCP), page 26

Exchanging and Understanding Content

The term **hypertext** was coined in the 1960s and was later extended to **hypermedia** to include sound, video, and other ways of presenting information.

Sir Tim Berners Lee later combined several technologies to create the infrastructure on which the web operates today:

- A language that allowed users to write hypertext documents (**HyperText Markup Language**, or **HTML**).
- A protocol to send those documents over the Internet when a link was followed (**HyperText Transfer Protocol**, or **HTTP**).
- These are both **public standards**, allowing anyone to publish web content.

XML and HTML

XML (eXtensible Markup Language) and **HTML (HyperText Markup Language)** are both markup languages for the representation of information. A markup languages provides **annotations** for text to denote **structure** and **display**.

See more: Mark-Up Languages, page 44

Security and Integrity

Public and Private Keys

Public/private key encryption can be used to send encrypted information **without ever sharing private information** or reserved secrets in advance.

Digital Certificates

Digital certificates are used to prove that content has come from the host it claims to be from (commonly used for websites). A popular certificate format is **X.509**, which contains three parts:

- The certificate details
  - Serial number, validity period, issuer and owner details, and the public key of the owner.
- The signature of the certificate.
- The algorithm used to sign the certificate.

Web Services

Web services can consist of a **private implementation** fronted by a **public interface** that uses the **Web Service Definition Language (WSDL)**.

The **Simple Object Access Protocol (SOAP)** can be used to exchange requests and responses. A request consists of an **HTTP** header and an **XML** message conforming to the **WSDL** interface.

See more: Web Services, page 55

Semantic Web

The goal of semantic web is to make content **machine-understandable**. It requires using **app-specific mark** up in web pages and creating **agreements on mark-up concepts and practises** amongst distributed users.

An agreed set of **concepts and meanings** in a **parsable** form is an **ontology**.

Resource Description Framework (RDF) is an XML-based specification to describe a particular resource. A set of RDF statements can form an **RDF graph** that contains values and resources at its nodes, and predicates (‘knows’, ‘controls’, ‘owns’, ‘observes’, etc.) along its edges.

Paradigm Shifts

Software based systems enable very quick modifications.

Cloud-based services are changing the ways people and industries consume technology.

- **SaaS (Software as a Service)**: whole applications can be rented or subscribed to (such as SalesForce CRM).
- **PaaS (Platform as a Service)**: platforms can be rented on which custom software can be deployed (such as Google Cloud Apps and Microsoft Azure).
- **IaaS (Infrastructure as a Service)**: processing and storage capacity can be rented (such as AWS and Rackspace).

Internet Architecture

Two key questions are addressed by the Internet's architecture:

- The Internet is huge - how can administration be divided into manageable chunks?
- The Internet is distributed - how can changes be implemented without breaking things or requiring changes elsewhere? (i.e. low blast-radius changes)

Architecture Goals

- **Connect** existing networks together.
- Be **robust** with regards to small-scale (individual links) and large-scale (entire subnetworks) failures.
  - Routing functionality should adapt to these situations.
- Allow **distributed management**.
- Support multiple types of content and service.
- Allow **host attachment** with little effort.
- Be **cost-effective** in terms of header overhead, re-transmission, required router capabilities, etc.

Architecture Layers

The Internet is organised as a set of layers. Many issues must be solved for a successful Internet application (routing, reliability, data formatting, flow control, etc.); **each layer solves one or a few** of these issues, and most layers have **multiple implementations**. This allows for different combinations of technologies to be selected to best suit a particular problem.

The main layers are, from the bottom up:

- **Physical** - the actual connectivity (copper, fibre, radio, etc.).
- **Access** - defines how to deliver data between two devices on the same network (most commonly Ethernet).
- **Network** - defines how to route messages across networks.
- **Transport** - defines how to provide reliable communication, so that data will not be lost or corrupted.
- **Application** - defines how programs instruct messages to be sent by the lower layers (encryption, compression, etc.).

This course focuses on the top three layers (network, transport and application).

Protocols

A protocol is a way of communicating. It specifies how to **express information**, how to **respond** when given certain requests or commands, and the **forms of requests or commands** to expect.

Each layer can be implemented by **multiple alternative protocols** that **guide the communication of hosts** to achieve the layer's purpose.

Network Layer

- **Internet Protocol (IP)** is the main protocol that is used.
  - IP can be used for transferring messages between hosts anywhere on the Internet.
  - *See more: Internet Protocol (IP), page 19*
- **Internet Control Message Protocol (ICMP)** is also sometimes used to augment IP.
  - *See more: Internet Control Message Protocol (ICMP), page 25*

Transport Layer

- **Transmission Control Protocol (TCP)**
  - Provides reliability measures (acknowledgements and flow control), sessions (container for multiple communications), multiplexing (bundling communications for multiple applications into one transmission)
  - *See more: Transmission Control Protocol (TCP), page 26*
- **User Datagram Protocol (UDP)**
  - Minimal overhead.
  - Some reliability measures provided by checksums, but otherwise unreliable.

Process/Application Layer

- HyperText Transfer Protocol (HTTP) (*see more: Hyper-Text Transfer Protocol (HTTP), page 36*)
- TELNET
- Simple Mail Transfer Protocol (SMTP)
- File Transfer Protocol (FTP)
- Post Office Protocol (POP)
- Domain Name Service (DNS)
- Dynamic Host Configuration Protocol (DHCP)
- etc.

Edge-Oriented Architecture

The Internet's success is due to its edge-oriented approach to architecture: a **connectionless, packet-forwarding infrastructure** (dumb network) that positioned **higher-level functionality at the edge** of the network for robustness.

Intelligent edges and a dumb network **keep the infrastructure as simple as possible**. Complexity of the core network is reduced, and new applications can be easily added.

Addresses in this system use **fixed sized numerical values** with **simple structures**. They are applied to physical network interfaces, so they can be used for **naming** a node and **routing** to it. *See more: Addressing, page 13*.

Packet Transmission

```
HTTP > TCP > IP > Link Layer > Copper
```

- HTTP encodes the message of data
- TCP adds its header, packet number, timeout settings, etc.
- IP adds host and destination information, routing information, etc.

IP Fragmentation

Different access layer technologies can carry **packets of different sizes**. The maximum packet size is called the **Max Transfer Unit (MTU)**. IP is encapsulated in the access layer, so the MTU of a particular access layer implementation **limits the size of IP packets** that can be sent through it.

If the outbound link has a smaller MTU than the IP packet the router wants to send, **fragmentation** is the solution: the packet is broken up, each fragment is sent, and the receiver re-assembles them.

See more: Fragmentation, page 19

Addressing

Key questions:

- How can hosts identify each other when they are not directly connected?
- How can addressing schemes handle various numbers of hosts in an organisation?

Networks

- For two hosts to communicate, there must be a connection between them (cable, wireless, etc.).
- A network is a set of computers **connected directly or indirectly**.
  - A computer that is part of a network is called a **node**.
  - A node from which messages are sent and/or received is called a **host**.
  - Other kinds of nodes are **routers**.

Routing and Addresses

Generally, one host wants to communicate with another host that it is **not directly connected to**. We need routing to achieve this:

- A path is found along a series of connected nodes.
- Data is sent along the resulting path until it reaches the destination (or fails).

How can a sender identify which receiver it needs to send to, so that a route can be found? **IP Addresses**.

IP Addresses

In the global Internet, each and every host and router needs **one globally unique address**. Technically, IP addresses are associated with a **network interface within a machine**, not a host.

Primarily **IPv4** is used; **IPv6** is being deployed slowly.

Both types of IP addresses use a **hierarchical structure**:

- The Internet is divided into networks.
- Each network has a network prefix.
- Each host in the network has a host identifier.

Together these make the IP address.

Network Prefixes

Global routers pass each message down to the **local network router(s)** for the given network prefix, which then passes the message to the host specified by the host ID.

Network prefixes assigned by **ICANN (Internet Corporation for Assigned Names and Numbers)** and **NICs (Network Information Centres)**. Owners of the prefixes assign host identifiers within them.

Some network prefixes are guaranteed not to be allocated, such as those used for technical purposes, internal networking (**`192.168.x.x`**), etc.

The whole address is always passed when routing a packet, so in order to determine which parts are the network prefix and host ID, routers need to know how long the prefix is. The length of a prefix in an address is indicated with a slash and the length, for example `143.326.3.26`**`/16`** for a **16-bit** prefix.

IPv4 Addressing

IPv4 addresses are **32 bits long**, giving $2^{32} \approx 4.3bn$ addresses. Within any network, two addresses are **reserved**:

- The prefix followed by **all 0s** (binary) - this is the address of **the network itself**.
- The prefix followed by **all 1s** (binary) - this is the network's **broadcast address**.

For example, if the network prefix is 23 bits long then there are 9 left for the host ID. The network can therefore hold $2^9 - 2 = 510$ hosts. (The -2 comes from the reserved all-0 and all-1 addresses.)

Network Prefix Classes

To provide flexibility to support **different network sizes**, three different classes of addressing were created (**A**, **B** and **C**), plus two non-standard classes for multicasting (**Class D**) and experimentation (**Class E**).

The class of an address and the subnet mask determine how many of the 32 bits belong to the network prefix and how many belong to the host ID.

- **Class A (/8)**
  - Used for very large networks.
  - Binary IP starts with `0...`
  - 8-bit network prefix, giving $2^7 - 2 = 126$ possible /8 networks.
    * 2^7 and not 2^8 because the first bit is always `0`.
    * -2 because `0.0.0.0` and `127.0.0.0` are reserved for the default route and local loopback functions.

  - 24-bit host ID, $2^{24} - 2 = 16,777,214$ hosts per network.
    - -2 because the all-0 and all-1 addresses are reserved.
  - Decimal address range 1 to 126.

- **Class B (/16)**
  - Used for large networks.
  - Binary IP starts with `10...`
  - 16-bit network prefix, giving $2^{14} = 16,384$ possible /16 networks.
    - 2^{14} and not 2^{16} because the first 2 bits are always `10`.
  - 16-bit host ID, $2^{16} - 2 = 65,534$ hosts per network.
    - -2 because the all-0 and all-1 addresses are reserved.
  - Decimal address range 128 to 191.

- **Class C (/24)**
  - Used for smaller networks.
  - Binary IP starts with `110...`
  - 24-bit network prefix, giving $2^{21} = 2,097,152$ possible /24 networks.
    - 2^{21} and not 2^{24} because the first 3 bits are always `110`.
  - 8-bit host ID, $2^{8} - 2 = 254$ hosts per network.
    - -2 because the all-0 and all-1 addresses are reserved.
  - Decimal address range 192 to 223.

Classful vs. Classless Addressing

Classful addressing has huge gaps between class sizes, so in 1993 **Classless Inter-Domain Routing (CIDR)** was standardised by the IETF. In CIDR-ised networks, the **network prefix can be any number of bits long**.

For example, to serve 2000 hosts, addresses in the form **`a.b.c.d`**`/21` could be assigned to leave 11 host ID bits, giving $2^{11} - 2 = 2046$ hosts.

Today, **address classes are ignored** and routers are explicitly told the prefix length.

Subnets

Subnets **split up a network** to give finer control and separation. With subnets, addresses take on a three-level structure of **network prefix**, **subnet ID**, **host ID**.

Subnet Masks

The subnet mask is used to separate the network prefix and the host ID. In binary format, it uses **1s to represent the network number** and **0s to represent the host number**.

For example, for a `/8` network the subnet mask would be `1111 1111 0000 ... 0000`.

- Prefix = `IP & Subnet Mask`
- Host = `IP & (~Subnet Mask)`

Subnet Example 1

An organisation has been assigned the network prefix `193.1.1.0/24` and wants 6 subnets for up to 25 hosts each.

- Network prefix: 24 bits.
  - Mask (bin): `11111111.11111111.11111111.00000000`
  - Mask (dec): `255.255.255.0`
- 3 bits are needed to define 6 subnets, so the extended network prefix has 27 bits.
  - Mask (bin): `11111111.11111111.11111111.11100000`
  - Mask (dec): `255.255.255.224`
  - This allows $2^3 = 8$ subnets, so there are 2 available for future growth.
- 5 bits are left for the host ID.
  - $2^5 - 2 = 30$ possible hosts.

Subnet Example 2

An organisation has been assigned 140.25.0.16/16 and needs to create subnets to support up to 60 hosts each.

- To define 60 hosts, the host ID needs 6 bits ($2^6 - 2 = 62$).
  - This is tight, so 7 bits are selected to give $2^7 - 2 = 126$ hosts per subnet.
- The network prefix has 16 bits and the host ID has 7, leaving 9 bits for the subnet ID.
- The extended network prefix is now $16 + 9 = 25$ bits long.
  - Mask (bin): `11111111.11111111.11111111.10000000`
  - Mask (dec): `255.255.255.128`
  - This allows $2^9 = 256$ subnets.

Variable-Length Subnets

Fixed-length subnets create problems. An organisation might need many subnets, but as the subnet ID grows, the number of possible hosts shrinks. They may also need subnets of different sizes.

Using **variable subnet ID lengths**, the host ID space can be iteratively divided into large blocks first, then smaller ones. As the number of bits used for the subnet ID varies, so too must the subnet masks.

Variable-Length Subnet Example

The network `a.b.c.0/24` needs the following five subnets:

- Subnet A requires 90 hosts.
- Subnet B requires 36 hosts.
- Subnets C, D and E require 12 hosts each.

A `/24` network can accommodate $2^{32-24} - 2 = 2^8 - 2 = 254$ hosts, so there is enough space!

- First, we can use 1 bit to split A from the rest of the address space.
  - Subnet A has a `/25` address.
  - `a.b.c.0xxxxxxx`
- Then the second largest (B) needs another bit to be separated.
  - Subnet B has a `/26` address.
  - `a.b.c.10xxxxxx`
- Finally the smaller subnets (C, D and E) need two more bits to be separated.
  - Subnets C, D and E have `/28` addresses.
  - `a.b.c.1100xxxx`
  - `a.b.c.1101xxxx`
  - `a.b.c.1110xxxx`
  - `a.b.c.1111xxxx` (spare)

IPv6 Addressing

IPv6 introduces several improvements on the IPv4 standard:

- Increased address space ($2^{128} \approx 2.3 * 10^{38}$ addresses).
- Network-layer encryption and other security features.
- Better flow control for better end-to-end service quality.
- Supports new features for new applications.

Addresses are 4 times as long as IPv4 (128 vs. 32 bits), but the header is only twice the size. Addresses are expressed in **8-word hex statements** with the same prefix-length notation (e.g. `2001:0db8:0000:0042:0000:8a2e:0370:7334/64`). All local IPv6 networks are /64.

Three types of IPv6 address exist:

- Unicast - single interface.

- Anycast - any host in a network.
- Multicast - every host in a network.

There are no address classes like IPv4, but two prefixes are reserved:

- `1111 1111 ...` is used for multicast.
- `1111 1110 10...` is used for link-local unicast.

Two addresses are reserved:

- `0::0` means 'the host has not been assigned an address'.
- `0::1` is used for loopback (for a host to send messages to itself).

Internet Protocol (IP)

IP is the **network layer** for the Internet - a host-to-host packet delivery service.

The key challenges for IP are:

- How can we send a message to the right destination?
- How can we send messages larger than some networks are able to handle?
- How can we know which protocols were used to send the message, so that we can interpret it correctly?

There are several issues that IP does not solve:

- It is **unreliable**.
- Messages may get **corrupted in transit**.
- Message fragments may arrive out of order, arrive duplicated, or not arrive at all.

Higher-level protocols like **TCP** add reliability to IP (*see more: Transmission Control Protocol (TCP), page 26*).

Fragmentation

Every physical network has a limit to the maximum message size it can transmit: it's **Maximum Transfer Unit (MTU)**. To work around this, any message can be **split into fragments** by the sender, each of which can be sent individually. Fragments are reassembled by the receiver (usually by the **TCP/IP** network driver).

IP adds a header to every fragment. Some fragments may take different routes to the destination. In IPv4, **fragments may be further fragmented** when passed to a network with a lower MTU.

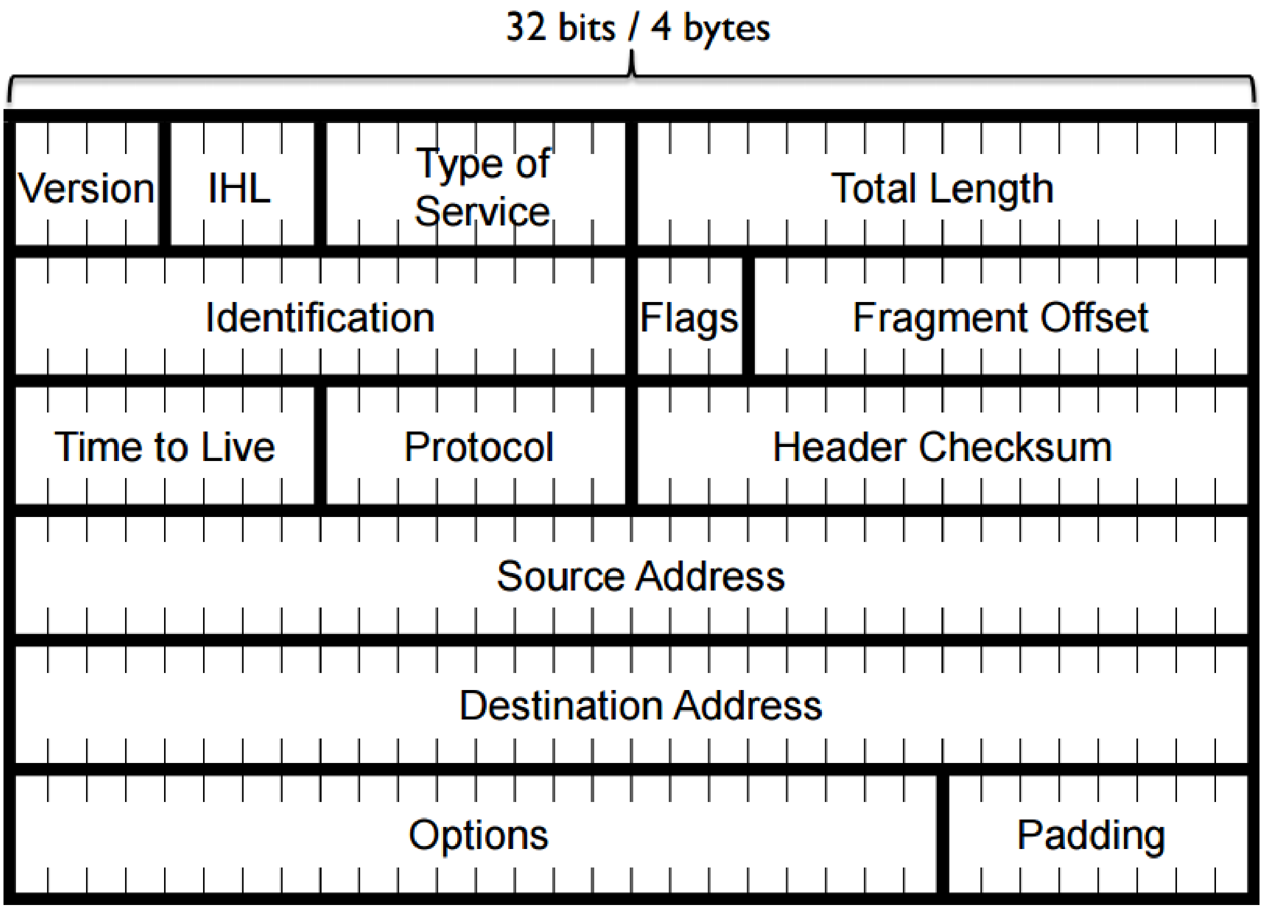
All fragments must be multiples of **64 bits (8 bytes)**, except the last one.

In general, fragmentation is a bad thing: it adds significant overhead (more header data) and delays (fragmentation/reassembly). It is necessary so that multiple networks can connect to an **open Internet**, regardless of their physical restrictions (MTUs).

See more: Path MTU, page 24

IPv4 Header

The IPv4 header consists of 5 32-bit (4-byte) words, with more 32-bit words occasionally added to specify options.



- Version (4 bits)
  - `0100` or `0110` for IPv4 or IPv6.
  - The same is used at the start of the IPv6 header.
- Internet Header Length (IHL) (4 bits)
  - Specifies how many 32-bit words are in the header.
  - Minimum value of 5; larger when options are used.
- Type of Service (8 bits)
  - Originally used for specifying the type of service required (to favour throughput vs. reliability).
  - Redefined by modern protocols for congestion handling.
- Total Length (16 bits)
  - Total length **in bytes** of the packet/fragment, **including the header**.
  - Minimum value of 20 (for the minimum 5-word header).
- Identification (16 bits)
  - ID for this message.
  - Every fragment with the same ID is part of the same message.
- Flags (3 bits)
  1. Reserved, always zero

2. **Don't Fragment (DF)**: tells the router not to fragment this packet. If the packet exceeds the MTU and DF is set, the packet is dropped and ICMP (*see more: Internet Control Message Protocol (ICMP), page 25*) is used to send an error message.
3. **More Fragments (MF)**: specifies that there are more fragments from the same message following this one.

- Fragment Offset (FO) (13 bits)
  - Specifies where this fragment fits into the original message.
  - Measured in the number 8-byte chunks that go before this fragment.
    - E.g. FO = 3 means that this fragment starts $3 * 8 = 24$ bytes into the message.
  - Allows values up to 8191.
- Time to Live (TTL) (8 bits)
  - Specifies how long this packet can remain in the system before reaching the receiver.
  - Every host must reduce this counter by 1 when routing the packet.
  - Packets are **dropped when TTL = 0**. This stop packets from getting stuck in loops.
- Protocol (8 bits)
  - Specifies which **transport layer** is being used to send messages via IP.
  - Main protocols: TCP = 6; UDP = 17; ICMP = 1.
  - Protocol IDs are **shared between IPv4 and IPv6**.
  - IDs are assigned by the Internet Assigned Numbers Authority (IANA), part of the Internet Corporation for Assigned Names and Numbers (ICANN).
- Header Checksum (16 bits)
  - IP does nothing to prevent corruption, so the checksum allows higher-level protocols to verify the **integrity** of a packet header.
  - The checksum must be updated by any node that changes the header (such as updating TTL).
  - *See more: Header Checksum, page 22*.
- Source/Destination Addresses (32 bits each)
  - IP addresses of sender and receiver.
  - *See more: IPv4 Addressing, page 14*.
- Options and Padding (32-bit words)
  - Optional arguments and flags used by IP processing software.
  - **Variable number of bits**, but always padded to 32-bit words with zeros.
  - Covers options for routing, tracing, etc., but **not used very often**.

Header Checksum

The header checksum verifies the **integrity** of a header, but **not authenticity** (i.e. it is for corruption detection, not security).

It is computed as follows:

- The header (excluding the checksum) is considered as a series of 16-bit words.
- The one's-compliment sum of the words is computed.
- The one's-compliment of that sum is computed - this is the checksum.

This example assumes that all words apart from the last one have already been summed:

One's-C sum of other words:	`....1100 1010 0101 1001`
Last word:	`+...0100 0101 1000 0000`
Sum result:	`= 1 0000 1111 1101 1001`
One's-C 'swing around':	`+.....................1`
One's-C sum:	`....0000 1111 1101 1010`
One's-C (checksum):	`....1111 0000 0010 0101`

To verify a header, the one's-compliment sum of all 16-bit words (including the header) is computed - if the header is not corrupted, this value will be zero.

Effects of Fragmentation

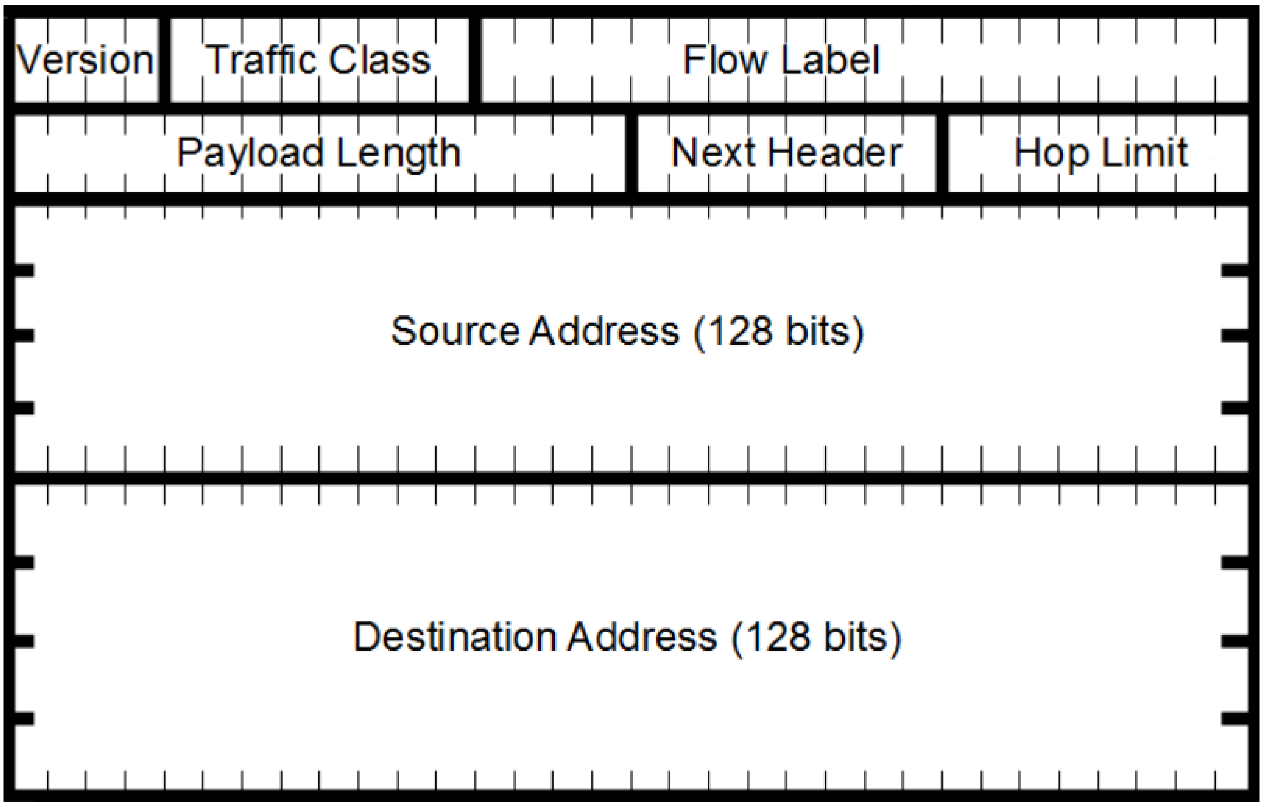
When a packet is fragmented, some **header values must change**.

- 'Total length' will change.
- The **MF** flag will be set to `1` for all fragments except the last one, which will keep its original value.
- The **FO** will change for all fragments except the first one.
  - The new FO for each fragment will be the original packet's FO, plus the fragment's offset (in 8-byte chunks) from the start of the original packet.

Note: fragments must always remain in multiples of 64 bits (8 bytes), except the last one.

IPv6 Header

Almost everything changes for the IPv6 header; **'version' is the only field that does not change**. IHL, flags, FO, header checksum and options/padding are removed entirely; everything else changes it's name and meaning. IPv6 headers are fixed at **80 bytes**.



- Traffic Class (8 bits)
  - Similar to 'Type of Service' in IPv4.
- Flow Label (20 bits)
  - Similar to 'Identification' in IPv4.
- Payload Length (16 bits)
  - Similar to 'Total length' in IPv4, but **excludes the header(s)**.
- Next Header (8 bits)
  - Similar to 'Protocol' in IPv4.
  - Specifies the next header on the packet (works like a singly-linked list).
  - *See more: Extension Headers, page 23*.
- Hop Limit (8 bits)
  - Similar to 'TTL' in IPv4.
- Source/Destination Addresses (128 bits each)
  - As before, just bigger.
  - *See more: IPv6 Addressing, page 17*

Extension Headers

IPv6 allows for options, but not directly inside the header. The main header can be followed by **zero or more extension headers**, all in a similar format to the IPv6 header format (but replacing addresses with their own protocol-specific information), **chained together with the ‘Next header’ field**.

Headers **specify a header protocol number** on the ‘Next protocol’ field to indicate that another 80-byte header follows it; the last header (which might be the first one) specifies a **higher-level protocol number to end the chain** and start the payload. Higher-level protocol numbers include 6 = TCP, 17 = UDP, 1 = ICMP, etc.

Examples:

IPv6 Header, Next header: TCP	TCP Header & Body

IPv6 Header, Next header: Routing	Routing Header, Next header: TCP	TCP Header & Body

IPv6 Header, Next header: Routing	Routing Header, Next header: Fragment	Fragment Header, Next header: TCP	TCP Header & Body

Fragment Headers

As shown above, fragmentation data is stored in extension headers in IPv6. These work the same way as in IPv4 and are only included when the message has been fragmented.

Path MTU

IPv6 **does not use fragmentation in transport**: it requires the sender to ensure messages are **sufficiently fragmented** to cross the network before sending them.

This is done by fragmenting messages according to the **path MTU**: the minimum MTU along the path from the sender to the receiver.

Note: the path MTU can be used with IPv4, but has to be implemented by a higher protocol like TCP (this is what the DF flag can be used for).

Discovery Algorithm

1. Assume the path MTU is the MTU of the first hop in the path (the link to the first router).
2. The sender fragments the message to the current assumed path MTU and sends the first fragment.
3. If the fragment reaches a link where it exceeds the MTU:
   (a) An ICMP ‘fragmentation needed’ error is send back to the sender, containing the lower MTU.
   (b) The sender updates the assumed path MTU to this lower value.

   (c) Go back to step 2.

4. When the first fragment reaches the destination, the path MTU is known and the rest of the message is sent.

IPv4 Don't Fragment (DF) Flag

If a fragment with the DF flag set reaches a link with an MTU lower than its size, an ICMP error is sent back to the sender (**ICMP code 4: 'fragmentation needed'**).

Internet Control Message Protocol (ICMP)

This protocol sits on top of IP and is used to report **error messages**, **routing information** and other IP processing messages back to the sender.

ICMP messages include a **message type and payload** where applicable. Some example message types:

- Destination unreachable error (payload specifies which hop failed).
- Echo request/echo response (used for ping).
- Redirection.
- Time exceeded.
- Router advertisement and router solicitation.

Transmission Control Protocol (TCP)

Many protocols are **encapsulated within IP datagrams** (via an inner header in IPv4 or extension header in IPv6) - TCP is one of them. TCP's main features are:

- **Reliability** - TCP guarantees delivery via acknowledgement and re-tries.
- **Multiplexing** - two hosts can have multiple 'conversations' without getting confused over which messages belong to which.
- **Flow/congestion control**.

Note: generally different protocols aim for **clean separation between layers** (*see more: Architecture Layers, page 10*), but with TCP this isn't quite the case, because its checksum (*see more: Header Checksums, page 33*) uses components of the IP header.

TCP Connections

Ports

TCP conceptually **divides a host's network interface into ports**, each of which can hold a separate channel of conversation. This is how **multiplexing** is achieved.

Some ports are reserved:

- Port 20/21 - FTP
- Port 25 - SMTP
- Port 80 - HTTP
- Port 443 - HTTPS

Sockets

A socket is a combination of a host's **IP address and port number**. Every TCP connection is between two sockets (i.e. two hosts using specific ports).

Initially, **a server will listen** on a given socket (e.g. a web server listening on port 80). **Clients can initiate** a connection to the socket offered by the server. The connection is usually initiated by something at application level, like a web browser.

Multiple clients can connect to the **same server socket** from different client sockets (such as many different web browsers connecting to the same server).

Once a pair of sockets is connected, data can be **sent in both directions** between the client and server (i.e. the connections are **full-duplex**).

Note: Clients and Servers

A **server** in this context is a host that is ready to **receive requests** and send data (later referred to as a **sender**). It signals to its TCP software that is ready to accept connections by sending a **passive OPEN request**.

A **client** in this context is a host that is **sending requests** and receiving data (later referred to as a **receiver**). It initiates a connection by sending an **active OPEN request** to a server.

Note: both roles could be filled by the the same host (such as a web browsers sending requests to a server running on `localhost`).

Connections

In TCP, hosts must establish a connection, requiring **set-up** and **tear-down**. Data can only be sent between hosts within a connection. There are a few reasons for this:

- It allows 'extra' information to be shared between hosts.
- It enables reliability.
- It allows resource reservation to ensure quality of service (more applicable on the server-end of the connection).
- It allows for flow control and congestion management.

A TCP connection is a kind of **session**; many other protocols use sessions as well.

Connection Set-up

A **3-part handshake sequence** of messages between a client and server is required to set up a connection before sending data.

1. The client sends a `SYN` (**synchronisation**) message to the server.
2. The server replies with a `SYN ACK` message to acknowledge the first message.
3. The client replies with an `ACK` message to acknowledge the acknowledgement.

Connection Teardown

A **3-part handshake sequence** is also used to close a TCP connection.

1. The sender sends a `FIN` message to **finalise** the connection.
2. The receiver replies with a `FIN ACK` message to acknowledge the first message.
3. The sender replies with an `ACK` message to acknowledge the acknowledgement.

Connections can be **closed from either side**. Connections in which only one end point has closed are in a **half-closed state**.

Sequence Summary

Client	**server**
`SYN` →	
	← `SYN ACK`
`ACK` →	
data →	
	← `ACK`
	← *data*
`ACK` →	
...	
`FIN` →	
	← `FIN ACK`
`ACK` →	

Flow Control

A host can only receive and process data at a given rate, which **may vary** depending on its processing load. If the rate of arriving data is too fast then eventually buffers will fill up and either **existing data will be overwritten** or newly **arriving data will be dropped**.

This is resolved within TCP by **allowing a receiver to tell the sender how much data it can handle**. The sender can then control the rate at which it sends the data to suit the receiver.

Segmentation and Acknowledgement

One part of the solution to flow control is **segmentation: splitting the message** to be sent into multiple segments, each of which can be transmitted separately.

This is similar to IP fragmentation (*see more: Fragmentation, page 19*), but at a higher level and for different reasons. IP splits messages to cope with the physical limits of network access layer protocols; TCP splits messages to cope with the limits of the receiving host, for flow control, and to help with reliability.

IP fragmentation is expensive because dropped segments will be re-tried, so **TCP tries to choose a segment size to match the path MTU** (*see more: Path MTU, page 24*).

Sequence Numbers

Every byte within a message from a client to a server has a **unique sequence number**, which is **used to re-assemble** the message from its segments.

For any given connection, there will be an **Initial Sequence Number (ISN)**, used as a point of reference for all bytes within the connection. The ISN is determined during the set-up handshake (*see more: Connection Set-up, page 27*).

The first byte of real data that is sent will have a sequence number of $ISN + 1$, because the original `SYN` message contains an imaginary 1-byte payload. A segment will contain all of the contiguous bytes of data in the range of sequence numbers $ISN + a$ to $ISN + b$.

See more: TCP Header, page 31.

Reliability and Acknowledgements

The receiver of a message will **acknowledge every segment** that it receives, using the sequence number to identify bytes that have been received. The sender will use these acknowledgements to **re-try** any segments that it believes have been dropped.

An acknowledgement from the receiver to the sender states that the receiver has **received all of the data before a given sequence number**.

For example, if the receiver receives the segments with sequence numbers $[1..20]$ and $[21..30]$, it will send **31**. If the receiver receives the segments with sequence numbers $[1..20]$ and $[41..50]$, it will send **21**; the server will know that the segments with bytes 21 and onwards may have been dropped and should be re-tried.

Re-send Strategies

When will a sender know when to re-send a segment? Two approaches:

- **Time-out**: a sender may re-try a segment if it has not received an appropriate acknowledgement within a given time frame.
- **Repetition**: a sender may re-try a segment X if it receives acknowledgements suggesting that other segments are being received but X was dropped.
  - For example, if the receives segments $[1..20]$, $[31..40]$, $[41..60]$ and $[61..80]$ it may send the acknowledgement for 21 four times, which suggests that the segment starting at 21 is missing.

Some implementations may combine these methods.

Maximum Segment Size (MSS)

This is the **maximum amount of data that can be accepted by a receiver** at once - segments bigger than this will always be dropped.

The MSS is specified by each connection during the set-up handshake (*see more: Connection Set-up, page 27*). It may be different for each end of the connection (i.e. larger segments can travel in one direction but not the other).

Window Size

This is the **maximum amount of data that can be processed by a receiver** at once (often informed by a combination of buffer sizes and processing speed).

This size can be **adjusted throughout the connection lifespan** to accept more/less data (achieved through messages sent to the sender).

Segment Sizing Problem

The amount of data that a receiver can accept depends on the rate at which it can process already-received data (i.e. its **window may be partially filled already**).

The sender can only ever be sure that data for which it has received an acknowledgement was actually accepted by the receiver.

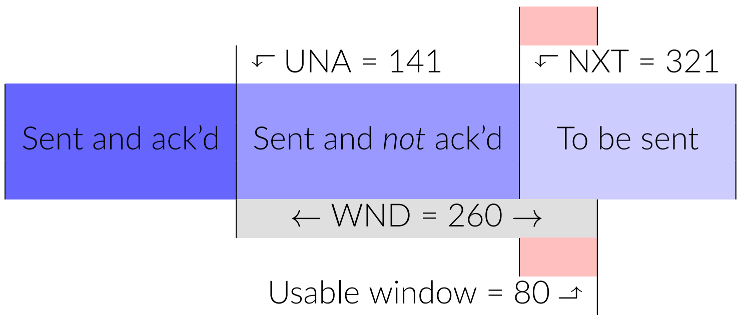
Solution: Usable Window Size

The usable window size is an estimate of the bytes that have **not yet been sent** but the sender believes the **receiver is ready to accept** (it is **computed by the sender**).

To determine it, the sender keeps track of **three variables**:

- UNA: the sequence number of the first byte that has been sent but not yet acknowledged.
- NXT: the sequence number of the next byte to be sent.
- WND: the window size reported by the receiver.

Usable window size: $UNA + WND - NXT$.



Silly Window Syndrome & Nagle's Algorithm

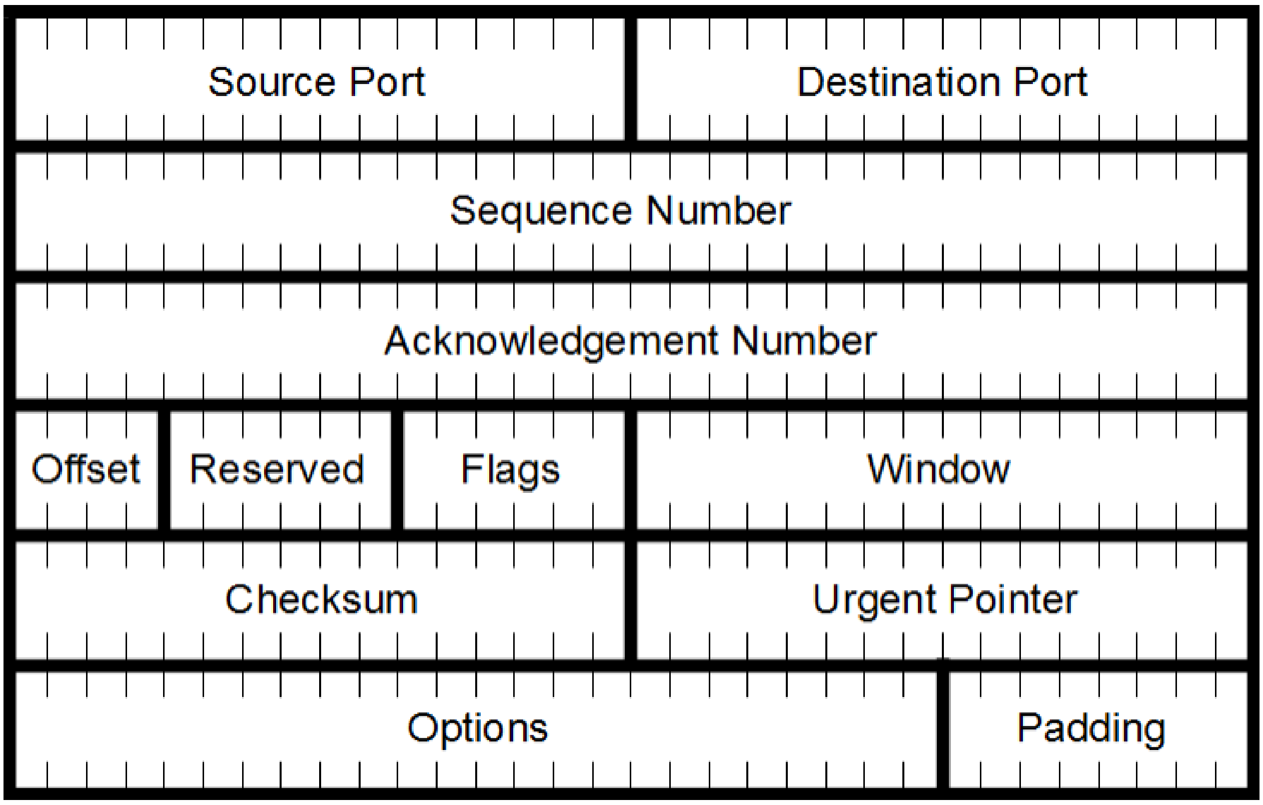
If the receiver adjusts the window size to be **too small**, bandwidth usage becomes **very inefficient**, because lots of very small segments will be sent, and there will be an acknowledgement for each. This is Silly Window Syndrome.

Nagle's algorithm is designed to help the sender and receiver work together to tackle this problem.

- A **sender** does not send more data until either...
  - ...all the data that has been sent as been acknowledged, or
  - ...the data to be send reaches the **MSS**.
- As it becomes able to accept more data, a **receiver** doesn't tell the sender about the larger window until either...
  - ...the window reaches the **MSS**, or
  - ...the window reaches half of the receiver's maximum buffer size.

TCP Header

Like the IPv4 header, the TCP header consists of 5 32-bit words, with additional 32-bit words sometimes used to specify options.



- Source/destination ports (16 bits each)
  - Unsigned integers (negative ports don't exist).

  - Source ports are allocated by TCP software.
  - Destination ports are chosen based on the service required.

- Sequence number (32 bits)
  - During the set-up handshake this is the initial sequence number (ISN).
  - In normal messages, this indicates the sequence number of the first byte of the segment.
  - *See more: Segmentation and Acknowledgement, page 28*.
  - *See more: Sequence Numbers, page 29*.

- Acknowledgement number (32 bits)
  - Used in `ACK` messages.
  - *See more: Segmentation and Acknowledgement, page 28*.

- (Data) offset (4 bits)
  - Length of the TCP **header only** in 32-bits (minimum value of 5).
  - Determines how far into the datagram the actual data starts.

- Reserved (6 bits)
  - Reserved for future use.

- Flags (6 bits)
  - Extra information about the message.
  - **`SYN`**: this is a set-up (synchronise) message.
  - **`FIN`**: this is a teardown (finalise) message.
  - **`ACK`**: this is an acknowledgement message.
  - **`PSH`**: historically this meant that all data that's ready to be sent should be sent right away; in practise this is now always set.
  - **`URG`**: indicates that this segment requires immediate action (useful on a slow connection) (example: `Ctrl + C` on a remote shell).
  - **`RST`**: *see more: Reset Flag, page 33*.
  - **Note**: flags can be mixed, for example a `SYN ACK` message.

- Window (16 bits)
  - The receiver's current window size.
  - Send in `ACK` messages.
  - *See more: Window Size, page 30*.

- Checksum (16 bits)
  - Used to validate the integrity of the header **and message**.
  - *See more: Header Checksums, page 33*.

- Urgent pointer (32 bits)
  - Used with the `URG` flag.

  - Points to the first byte **after** the urgent data in this segment.
  - Note: segments may mix urgent and non-urgent data.

- Options and Padding (32-bit words)
  - Optional arguments and flags used by TCP processing software.
  - **Variable number of bits**, but always padded to 32-bit words with zeros.
  - Example use: declaring the maximum segment size (MSS) during the set-up handshake.

Reset Flag

If one host **crashes** or there are **severe transmission problems**, one end of a connection may lose its knowledge of that connection, meaning it **will not be expecting the data** sent from the other end.

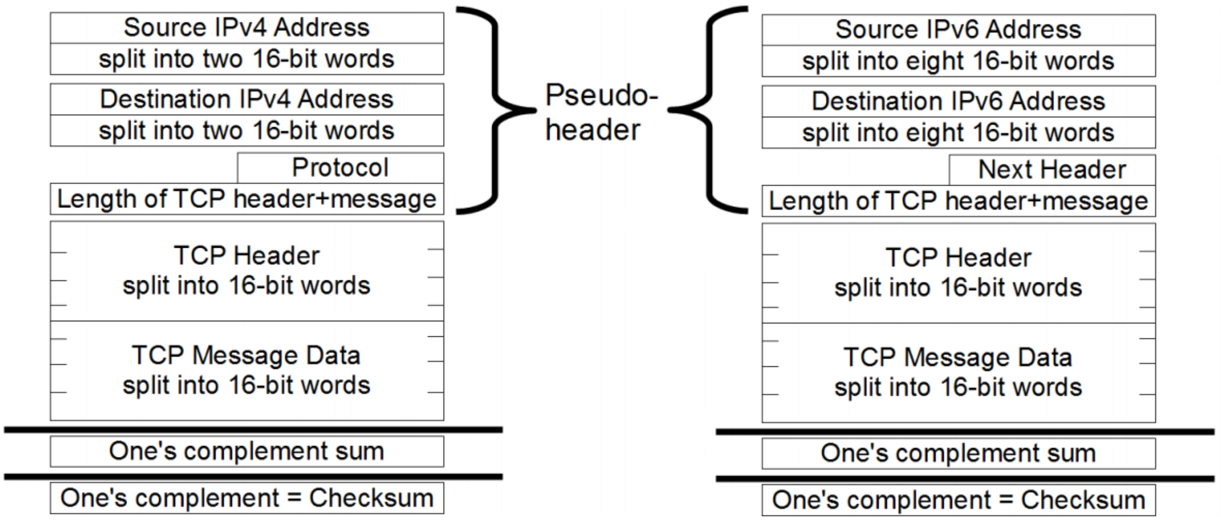
If a host receives unexpected TCP data it responds with a **reset** message to **stop to connection** and resolve the problem by starting the connection again.

The `RST` flag is also sent if a client tries to communicate with a **port that is not open**.

Header Checksums

As with IPv4 (*see more: Header Checksum, page 22*), the TCP header checksum is the one's-compliment of a one's-compliment sum of 16-bit words. The words included are:

- The TCP header (with the checksum set to zero).
- The message data.
- The IP addresses (from the IPv4/6 header).
- The protocol/next header field (from the IPv4/6 header).
- The length of the TCP message (header **and** data).
- The components of the IP header make up a section called the **pseudo-header**, shown on the following diagram for TCP/IPv4 and TCP/IPv6 checksums.



TCP and IP

TCP **runs over** IP: TCP datagrams become the **payload/content** of IP datagrams.

IP deals with addressing hosts and fragmentation to ensure the network can transmit the data.

TCP can more or less ignore what IP does and just pass data to send to a given address.

TCP and IP are **not fully separated**, because the TCP header checksum depends on parts of the IP header (*see more: Header Checksums, page 33*).

Full TCP Example

Data is sent via TCP between two hosts, both using port 9090. The data is 40 bytes long, split into two 20-byte segments. The client sets the ISN to 25. The server has a buffer size of 100 bytes and does not process data until after it is all received. No options or additional flags are set.

Client →	**← Server**
Set-up	
Seq.: 25 Offset: 5 Flags: `SYN` *Imaginary 1-byte payload*	
	Ack.: 26 Offset: 5 Flags: `SYN ACK` Window: 100
Offset: 5 Flags: `ACK`	
Data	
Seq.: 26 Offset: 5 Data: *first 20 bytes*	
	Ack.: 46 Flags: `ACK` Window: 80
Seq.: 46 Offset: 5 Data: *next 20 bytes*	
	Ack.: 66 Flags: `ACK` Window: 60
Teardown	
Offset: 5 Flags: `FIN`	
	Offset: 5 Flags: `FIN ACK`
Offset: 5 Flags: `ACK`	

Source/destination ports of 9090 would be present in every message.

Hyper-Text Transfer Protocol (HTTP)

The HTTP protocol is designed for communicating **documents and media** between computers. It is almost always **sent via TCP** (*see more: Transmission Control Protocol (TCP), page 26*), but could work with other protocols. The current version is **HTTP 1.1**.

Requests and Responses

HTTP uses a **client/server** model to exchange **requests** and **responses**.

- Clients send requests
  - The software sending the request is the **user agent**.
  - Usually a web browser, but may be a web crawler or other service.
- Servers respond to requests
  - Refers to an HTTP server, or any HTTP request-handling software in general.

Usually we have a TCP connection directly between the client and server, not this is not always the case: sometimes, **intermediate nodes** act to provide services such as translation, caching, firewalling, etc.

Proxies

A proxy is an intermediate node that **may transform a message en-route**. Clients know that they are using a proxy to communicate with a server. They are typically used for caching, privacy, etc.

Gateways

A gateway is an intermediate node that **just forwards requests** - sometimes used to solve routing problems caused by firewalls. The client talks to the gateway as if it is the final server.

Pipelining

For efficiency, when a client has **several requests** for a server, they can be pipelined and **sent through a single TCP connection** without waiting for a response in between each request.

The client distinguishes which responses match which requests based on their content.

Commonly used when loading a web page, to load the page, stylesheets, scripts, images, etc. at the same time.

Resources

HTTP is used to **retrieve and manipulate resources**, which are identified with a **Uniform Resource Identifier (URI)**. A resource might be a web page, a service, a file, a database, etc.

URIs can be **URNs** or **URLs**, or both.

- **Uniform Resource Name (URN):** the name for a resource, which is globally unique and should still have meaning after that resource ceases to exist (for example, the name of a person or ISBN of a book).
- **Uniform Resource Locator (URL):** a URI that can also be used to retrieve the resource named (these must specify a **method** and have a **means of retrieval**).

General URI Syntax

```
<scheme>:<scheme-specific details>
```

The first part determines how to interpret the second part. Possible schemes include `http`, `https`, `ftp`, `magnet`, etc.

HTTP URL Syntax

```
http://<host>[:<port>][<path>[?<query>]]
```

- `http` - denotes that this is a HTTP URL.
- `host` - domain or IP with the resource we want.
- `port` - TCP port.
- `path` - path to the resource, within the host.
- `query` - an optional query to send with the request.

Domain Name System (DNS)

IPs are hard to remember, so domain names are used as **human-readable aliases** for them. DNS **resolves** a domain name into an IP address.

Safe Characters and IRIs

Only a small set of characters can be used in the scheme-specific part of URIs:

```
A-Z a-z 0-9 $ - _ . + ! * ' ( )
```

All other characters must be encoded with `%` followed by their hexadecimal ASCII code (e.g. `%20` for the space character).

International Resource Identifiers (IRIs) were developed to allow non-English URIs.

URI Templates

URI templates are **compact representations** of a wide range of URIs, often used when configuring a web server. For example, the following covers all staff home pages in the Informatics department:

```
http://www.inf.kcl.ac.uk/staff/{username}/
```

HTTP Requests

Methods

All requests are typed by the method they perform:

- **`GET`** requests the retrieval of a resource.
- **`POST`** submits a resource.
- **`HEAD`** gets just the header of the response that `GET` would have returned.
- **`OPTIONS`** requests a list of the available communication options.
- **`PUT`** requests the storage of a resource.
- **`DELETE`** requests the deletion of a resource.
- **`TRACE`** requests that the request itself be returned (used for diagnostics).
- **`CONNECT`** is used for tunnelling HTTP requests through a secure proxy.

`GET`, `HEAD`, `OPTIONS` and `TRACE` are **safe methods**, because they do not request modifications. HTTP specifications state that these methods should do nothing more than return information.

Knowing that a method is non-safe allows requests to be confirmed before something unfixable is done.

Idempotency

Idempotent methods have the **same effect no matter how many times they are executed**. `GET`, `HEAD`, `OPTIONS`, `TRACE`, `PUT` and `DELETE` are usually considered to be idempotent, but `POST` is not.

Idempotent methods may be combined to create non-idempotent sequences. For example, alternations of `GET x` and `PUT x + 1` would have a different effect each time they are executed.

A client should make sure that non-idempotent methods and sequences are **not pipelined**. If the connection were to break, the client may not know how far it got through the pipelined sequence and so may repeat some requests.

Request Format

An HTTP request is a series of lines:

- Request line: the method, resource and HTTP version.
- General headers: data about the message transmission or protocol.
- Request headers: parameters of the request.
- Entity headers: describes the request's data (optional).
- Entity body: the data to send, if any (optional).

For example:

```
GET /index.html HTTP/1.1                                 Request line
Connection: close                                        General header
Host: example.net                                        Request header
Accept: text/html, text/plain                            Request header
User-Agent: Mozilla/4.0 (compatible; MSTE 6.0)           Request header
```

Response Format

An HTTP response is also a series of lines:

- Status line: the HTTP version, status code and reason phrase.
- General headers: data about the message transmission or protocol.
- Response headers: parameters of the response.
- Entity headers: describes the response's data.
- Entity body: the data to send, if any.

For example:

```
HTTP/1.1 200 OK                           Status line
Date: Wed, 26 Oct 2016 16:33:32 GMT+1     General header
Connection: close                         General header
Server: Apache/1.3.31                     Response header
Content-Type: text/html                   Entity header
Content-Length: 5474                      Entity header
                                          Blank line
<html>                                    Entity body
...                                       Entity body
```

Response Codes

HTTP response codes are 3-digit numbers representing whether or not a request was fulfilled correctly, and the specific error if it wasn't. They are usually accompanied by a human-readable **reason phrase**, and fall into 5 groups:

- `1xx` - information; no indication of success/failure.
- `2xx` - success; request was understood and accepted by the server.
- `3xx` - redirection; further action required.
- `4xx` - client error; invalid request.
- `5xx` - server error; unable to complete request.

Multipurpose Internet Mail Extensions (MIME)

HTTP and email were originally designed only for ASCII text, but now they carry various types of content. MIME specifies **how to declare the type of data being transferred** and allows **non-ASCII data to be encoded** for transmission over ASCII protocols.

MIME Types

MIME provides a high-level classification of data into 7 types:

- `text`
- `image`
- `audio`
- `video`
- `application`
- `multipart`
- `message`

New types must be preceded with ‘X-’ to denote them as experimental.

Each MIME type may have many sub-types, and a full media type is written as `<type>/<sub−type>`. Examples:

- `text/plain`
- `text/html`
- `image/jpeg`
- `audio/mp4`

Optional parameters can be added to give even more detail, for example:

```
text/plain; charset=ISO−8859−1
```

HTTP and MIME

The type of an entity in an HTTP response is defined by the **entity header**, as so:

```
Content−Type: text/html
```

The type of data accepted in response to a `GET` request is defined in the **request header**, as so:

```
Accept: text/plain, text/html
```

MIME Encoding

Some protocols have limits on the format and structure of data, so data has to be encoded to make it transferable. The encoding used must be declared, so that it can be decoded at the other end. HTTP headers allow for such declarations:

```
Content−Transfer−Encoding: <encoding>
```

MIME data can be encoded in 5 different ways:

- **7-bit** is ASCII-compatible, with lines up to 1000 characters.
- **8-bit**, with lines up to 1000 characters.
- **Binary**, which cannot be used for direct transmission in SMTP.
- **Quoted-printable**, which is non-standard ASCII text.
- **Base-64**, which is binary data encoded into ASCII.

Base-64 Encoding

Every 24 bits of data is split into 4 6-bit chunks, each of which could have 64 different values. Each chunk is turned into a letter, number or symbol according to the logic below:

- 00 - 25: A - Z
- 26 - 51: a - z
- 52 - 61: 0 - 9
- 62: +
- 63: /

Where data does not divide into 24 bit chunks, '=' is used to encode the missing chunks. For example:

- `48` in hex → `SA==` in base-64.
- `48, 65` in hex → `SGU=` in base-64.
- `48, 65, 6C` in hex → `SGVs` in base-64.

Web Servers

A web server can be a program, a pre-packaged software/hardware unit, or embedded into a consumer product (like a router).

A server's basic algorithm is as follows:

1. Set up TCP connection.
2. Receive HTTP request.
3. Process request.
4. Access resource referred to by request.
5. Construct HTTP response.
6. Send HTTP response.
7. If requested, close connection.
8. Log the transaction.

Virtual Hosts

Virtual hosts are host addresses that **appear to be different**, but just map to different (or sometimes the same) content roots or applications on the **same server**.

Virtual hosts can be name-based or IP-based:

- **Name-based**: multiple domain names resolve to the same IP (and therefore the same host). Each domain corresponds to a different virtual host, so has a different document root.
- **IP-based**: multiple IP addresses are routed to the same host, and any message sent to any of those IPs will go to the host. Each IP corresponds to a different virtual host, so has a different document root.

Mark-Up Languages

'Marking up' means adding annotations around text to **explicitly denote properties** of it. HTML/XML are examples: HTML for web page layouts, and XML as a more generalised form. A mark-up language is a format where text is given **computer-parseable** information on:

- How text should be interpreted.
- How text should be presented.
- Which sections of text relate to others.

eXtensible Mark-up Language (XML)

eXtensible Mark-Up Language (XML) is a general purpose format for arbitrary data, not just readable text documents.

Tags

Mark-up is denoted by **pairs of opening and closing tags** around the piece of text or data to be annotated: `<name>Mark</name>`.

The parts of a document surrounded by opening/closing tags are called **elements**, which can be nested in a hierarchy:

```
<person>
    <name>Mark</name>
    <age>23</name>
</person>
```

Some elements can be empty, where the mark-up alone has meaning without any data inside it. In HTML for example, `<hr />` denotes a horizontal rule across the page (*see more: Hyper-Text Mark-Up Language (HTML), page 51*).

Attributes

Attributes can be used to **add parameters** to tags and provide specific information about the elements they wrap: `<hr width="80%" />`.

All attributes take the format `name="value"`. The attributes `xmlns` and `xml:lang` are reserved to denote the namespace and language of the document (*see more: Namespaces, page 45*).

Documents

An XML document must contain a **single root element** that encloses all other elements. Preceding the root element should be two special tags:

- The XML declaration.
- An optional document type declaration.

The XML declaration contains the version and character set of the document, and whether it is a 'stand-alone' document that can be parsed on its own, or if it needs another document to be parsed first. The declaration takes the following form:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
```

Entities

As XML uses special characters to delimit elements and attributes, these characters cannot be used inside the marked-up text itself. Instead, entities (special string) are used in their place:

```
&quot; = "      &amp; = &
&lt; = <        &gt; = >
&apos; = '
```

Comments

These are not parsed, but used to help anyone looking at the XML directly.

```
<!-- comment -->
```

Namespaces

Each XML-based application will **understand certain tags and attributes**, relevant to its role or purpose. With XML being exchanged between hosts and between applications, it's important to **distinguish tags with different meanings** in different contexts. This is where namespaces are used.

Namespaces are used to allow software to know how to interpret tags. A namespace is **identified by a URI**, and every element and attribute must have a namespace. There are two ways to declare the namespace of a tag or attribute:

- The **default namespace** gives a namespace to everything in its hierarchy, unless overridden.
- **Namespace prefixes** are identifiers added to a single tag or attribute.

The `xmlns="NS—URI"` attribute within an element specifies the **default namespace** for the element, all elements within it, and all of their attributes. For example:

```
<book xmlns="http://xml.com/books">
    <title>Some title</title>
    <author xmlns="http://xml.com/people">
        <title>Mrs</title>
        <name>Jane Doe</name>
    </author>
</book>
```

Alternatively, prefixes can be defined with the `xmlns:PREFIX="NS—URI"` attribute:

```
<book:book
    xmlns:book="http://xml.com/books"
    xmlns:person="http://xml.com/people">

    <book:title>Some title</book:title>
    <book:author>
        <person:title>Mrs</person:title>
        <person:name>Jane Doe</person:name>
    </book:author>

</book>
```

XML Schema

XML schemas are themselves written in XML. An XML schema document is also referred to as an **XML schema definition** and given the file extension **.xsd**.

The root element is `<schema>`, which has attributes to define the namespace used in the document and the target namespace of the elements and attributes defined in the document:

Elements and types defined in the schema have the target namespace, therefore references to **elements and types must use this namespace**. The target namespace prefix will be assumed as `tns` for the rest of this section.

```
<xs:schema
    targetNamespace="http://xml.com/MY—SCHEMA"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://xml.com/MY—SCHEMA">

</xs:schema>
```

Simple Types

A simple type is a type for the text inside elements and for attribute values. There are pre-defined simple types, such as string, integer, boolean, decimal, time, date, dateTime, etc.

To assert in a schema that an **element** has a given simple type, this structure is used:

- Schema format: `<xs:element name="ELEMENT-NAME" type="TYPE" />`
- Schema example: `<xs:element name="age" type="xs:integer" />`
- Conforming XML: `<age>23</age>`

To assert in a schema that an **attribute** has a given simple type, this structure is used:

- Schema format: `<xs:attribute name="ATTR-NAME" type="TYPE" />`
- Schema example: `<xs:attribute name="width" type="xs:decimal" />`
- Conforming XML: `<hr width="80%">`

Elements and attributes can also be given default or fixed values:

```
<xs:element name="quantity" type="xs:integer" default="9001" />
<xs:element name="the-answer" type="xs:integer" fixed="42" />
```

Simple Enumerator Types

Custom simple types can be defined as an enumerator:

```
<xs:simpleType name="sizeType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="S" />
        <xs:enumeration value="M" />
        <xs:enumeration value="L" />
        <xs:enumeration value="XL" />
    </xs:restriction>
</xs:simpleType>

<xs:attribute name="shirtSize" type="tns:sizeType" />
```

Simple Pattern Types

Custom simple types can be defined as a pattern:

```
<xs:simpleType name="postCodeType">
    <xs:restriction base="xs:string">
        <xs:pattern value="[A-Z]{1,2}[0-9]{1,2}[A-Z]? [0-9][A-Z]{2}" />
    </xs:restriction>
</xs:simpleType>

<xs:attribute name="postCode" type="tns:postCodeType" />
```

Pattern types work like regular regex.

Embedded and Referenced Complex Types

To specify the **hierarchical structure of an element**, a complex type is used. Complex type definition have the following format:

```
<xs:element name="ELEM-NAME">
    <xs:complexType>
        ...
    </xs:complexType>
</xs:element>
```

Complex types can also be **named and referenced** by multiple elements:

```
<xs:complexType name="TYPE-NAME">
    ...
</xs:complexType>

<xs:element name="ELEM-NAME" type="tns:TYPE-NAME" />
```

Complex Types: Sequence

The `<sequence>` complex type requires **all** sub-elements to be present, in the **specified order**.

```
<xs:complexType name="TYPE-NAME">
    <xs:sequence>
        <xs:element ref="tns:title" />
        <xs:element ref="tns:section" />
    </xs:sequence>
</xs:complexType>
```

Complex Types: All

The `<all>` complex type requires **all** sub-elements to be present, but in **any order**.

```
<xs:complexType name="TYPE—NAME">
    <xs:all>
        <xs:element ref="tns:title" />
        <xs:element ref="tns:section" />
    </xs:all>
</xs:complexType>
```

Complex Types: Choice

The `<choice>` complex type requires **any one** sub-element to be present.

```
<xs:complexType name="TYPE—NAME">
    <xs:choice>
        <xs:element ref="tns:synopsis" />
        <xs:element ref="tns:blurb" />
    </xs:choice>
</xs:complexType>
```

Complex Types: Min/Max Occurrences

The `minOccurs` and `maxOccurs` attributes restrict the number of times that a sub-element can be present (both default to 1).

```
<!—— 0 or 1 ——>
<xs:element ref="tns:blurb" minOccurs="0" maxOccurs="1" />

<!—— 2+ ——>
<xs:element ref="tns:section" minOccurs="2" maxOccurs="unbounded" />
```

Complex Types: Combining Structures

Structures can be combined to create more detailed complex types.

```
<xs:element name="contact">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="name" type="xs:string" />
            <xs:choice>
                <xs:element
                    name="postAddress"
                    type="tns:postAddressType" />
                <xs:element
                    name="emailAddress"
                    type="tns:emailAddressType"
                    maxOccurs="unbounded" />
            </xs:choice>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Any Types

The `<any>` element can be used where the schema allows the XML to be extended with any arbitrary content. Three varieties exist:

- `<xs:any />` - any element or attribute.
- `<xs:anyElement />` - any element.
- `<xs:anyAttribute />` - any attribute.

Attributes of Elements

Attributes are defined **after sub-elements** in a complex type.

```
<xs:complexType name="TYPE—NAME">
    <xs:choice>
        <xs:element ref="tns:blurb" />
        <xs:element ref="tns:synopsis" />
    </xs:choice>
    <xs:attribute name="author" type="xs:string" />
</xs:complexType>
```

Interleaved Text

To interleave text and mark-up in an element (as in HTML bodies), we declare its type as `mixed`:

```
<xs:complexType name="TYPE—NAME" mixed="true">
    <xs:choice>
        <xs:element ref="tns:blurb" />
        <xs:element ref="tns:synopsis" />
    </xs:choice>
</xs:complexType>

<xs:element name="ELEM—NAME" type="tns:TYPE—NAME" />
```

Usage:

```
<ELEM—NAME>
    <synopsis>Stuff</synopsis>
    This is some text inside ELEM—NAME, not in any other mark—up.
</ELEM—NAME>
```

Hyper-Text Mark-Up Language (HTML)

Hyper-Text Mark-Up Language is the language used to encode web pages and other simple-format documents, like emails. It is an **XML-like** format with pre-defined elements and attributes to describe both the **structure of pages** and the **links between them**.

HTML is **non-linear** because it includes links to other text, so that text branches and can be read via multiple paths.

HTML and XHTML

HTML has gone through several version, the current latest being **HTML 5**.

HTML 4.01 was a near-XML structure, but with a few differences: namespaces meant nothing, and not all tags needed to be closed.

XHTML 1.0 is a fully-XML structure, but in reality the differences between it and HTML 4.01 are very small.

HTML 5 can be written in the form of HTML 4.01 or XHTML 1.0 and provides additional elements and attributes for embedding media, typesetting documents, form inputs, etc.

HTML Structure

The root tag for an HTML document is `<html>`, which usually contains two child tags, `<head>` and `<body>`. The head of a document is for non-displayed meta data, and the body defines the displayed content.

Text in the `<body>` section will be interpreted and displayed by a web browser, which decides *how* to display it (and ignores whitespace).

Structural Mark-Up

- `<h1..7>` tags specify a hierarchy of headings (1 being the highest).
- `<p>` tags specify paragraphs.
- HTMl 5 adds many more structuring tags, like `<section>`.

Presentational Mark-Up

- `<br />` adds a single line break.
- `<strong>` emboldens text.
- `<em>` adds emphasis to text (usually italics, but may vary depending on custom styles and/or browser preferences).

Lists

Un-ordered list (bullet points):

```
<ul>
    <li>List item 1</li>
    <li>List item 2</li>
</ul>
```

Ordered list (numbers):

```
<ol>
    <li>List item 1</li>
    <li>List item 2</li>
</ol>
```

Images

```
<img src="pics/something.png" />
```

The `src` URL is used by the web browser to issue a GET request for the image. The URL may be relative or absolute.

Tables

3 columns, 2 rows:

```
<table>
    <tr>
        <td>R1, C1</td>
        <td>R1, C2</td>
        <td>R1, C3</td>
    </tr>
    <tr>
        <td>R2, C1</td>
        <td>R2, C2</td>
        <td>R2, C3</td>
    </tr>
</table>
```

Tables may also contain `<thead>`, `<tbody>` and `<tfoot>` elements, each containing one or more rows, for further structuring. Cells inside `<thead>` should use `<th>` instead of `<td>`.

Links

The `<a>` tag is used for links, with the `href` attribute used to specify the destination.

```
<a href="http://lmgtfy.com?q=html">How to HTML</a>

<a href="/">Relative link to home</a>
```

Anchors can also be used to **link to other sections** within a single page:

```
<a name="faq" />

...

<a href="#faq">FAQ on this page</a>

<a href="http://something.com/page#faq">FAQ on another page</a>
```

Forms

HTML documents can have forms for user to **submit information**. On submission (usually triggered by a button click), the form data is sent to a HTTP server. A `<form>` element has two attributes that specify how and where the data is sent:

- `method` - the HTTP method to use for submission.
    - *See more: Methods, page 38*.
- `action` - the destination for submission.

A form can contain normal HTML, as well as `<input>` elements specifying components for the user to fill in and one or more buttons used to send or reset the form. Every input has a `name` and `type` attribute. An example form:

```
<form action="/login" method="post">
    <p>Username:</p>
    <input name="username" type="text" />

    <p>Password:</p>
    <input name="password" type="password" />

    <p>Stay logged in?</p>
    <input name="stay" type="checkbox" value="1" />

    <input type="submit" value="Login" />
</form>
```

Form data is submitted with the MIME type `application/x-www-form-urlencoded` (*see more: MIME Types, page 40*) and is encoded into a `name=value` format, with spaces and other non-URL-safe characters encoded (*see more: Safe Characters and IRIs, page 37*).

When forms are submitted with the GET method, the data is appended to the URL after a `?` symbol:

```
/login?username=mark&password=alligator3
```

When forms are submitted with the POST method, the data becomes the HTTP request's entity body (*see more: Request Format, page 39*).

Cascading Style Sheets (CSS)

Cascading style sheets are used to **define the appearance** of an HTML document and can be encoded into the `<head>` section or (preferably) included from a separate file (thus allowing re-use).

CSS allows for separation of content structure (semantics) and appearance (aesthetics).

Web Services

Service-Oriented Computing

This concept applies the principles behind the Internet and web to software functionality, including **decentralisation**, **distributed administration**, **consistent protocols** and provisions for companies to **provide, maintain and update proprietary software** on their own sites.

Multiple services may be combined to provide a complete produce.

Service-oriented computing is typically structured to follow **object/component** principles.

Interfaces

Services expose an interface that **defines the protocols that are supported**, some or all of which may be specific to the service or organisation providing it. Multiple services can be **interchanged** to provide the same functionality if they support the same interface definition.

Messages that conform to the interface are passed between client and service. This is achieved by **publishing the interface** and allows the **implementation to be private**.

Web Services

Web services are simply services that are deployed using the Internet and web technologies, such as communication via HTTP and communication in XML (*see more: Hyper-Text Transfer Protocol (HTTP), page 36*; *see more: eXtensible Mark-up Language (XML), page 44*).

Key technologies are:

- **SOAP**: an XML-based communication protocol.
- **WSDL**: an XML-based interface definition language.

Simple Object Access Protocol (SOAP)

SOAP is the web service communication protocol. Messages have a common structure with an **outer envelope** containing a **header** followed by a **body**.

Message Structure

The body contains **a message conforming to the service's interface definition**; the header contains information about the communication, the body, the sender, authentication, addressing, etc.

Both components are XML elements. The SOAP XML schema allows **any XML content** in the header and body.

An example structure is as follows:

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/03/soap-envelope">
    <soap:Header>
        <t:Transaction
            xmlns:t="http://example.com"
            soap:mustUnderstand="1">
            5
        </t:Transaction>
    </soap:Header>
    <soap:Body>
        <m:GetStockPrice xmlns:m="http://example.com/stock">
            <m:company>GOOG</m:company>
        </m:GetStockPrice>
    </soap:Body>
</soap:Envelope>
```

The `mustUnderstand` flag can carry a `1` or a `0`; with a `1` it indicates to the receiver that they must understand the meaning of this particular element, otherwise the whole message should be rejected.

SOAP Over HTTP

SOAP is commonly sent over HTTP with the envelope and its contents as the entity body in the HTTP request/response.

SOAP Actions

SOAP allows the **intent** of the message to be specified as a `SOAPAction` URI in the HTTP `Content-Type` field. It provides extra information about how to process the message.

```
POST /stock-quote HTTP/1.1
Host: example.com
Content-Type: application/soap;
              charset=utf-8;
              SOAPAction=http://example.com/GetStockPrice

<soap:Envelope
...
```

Web Service Definition Language (WSDL)

WSDL is an XML-based language for specifying the form of **messages accepted and generated** by a web service. It specifies the rules on the form of a SOAP body (i.e. a schema, written in XML schema; *see more: XML Schema, page 46*).

A service interface is split into **port types**, each of which contains a set of **operations** made up of **input and output message definitions**.

An **operation** combines an input and output message to say 'this form of input will generate this form of output'. An operation may accept multiple input formats.

Port Types and Operations

A port type groups a **set of operations** of a particular kind. For example, a registry service might have a 'publish' port for registration-related operations and an 'inquiry' port for search-related operations. A port type specifies which messages can be received and produced by a port.

An operation is something that can be **performed** on a service, like a method in OOP. Operations specify an input message definition and an output response definition. They are assumed to be **asynchronous**.

An example port type and operation:

```
<portType name="StockQuotePortType">
    <operation name="GetStockPrice">
        <input message="tns:GetStockPriceInput" />
        <output message="tns:GetStockPriceOutput" />
    </operation>
</portType>
```

Messages

Messages are XML documents (the contents of SOAP bodies). They must **conform to a schema**, so that services and clients known the expected format of requests and responses. One type of message could be the response for multiple operations.

An example message definition:

```xml
<schema>
    <element name="PriceRequest">
        <complexType>
            <all>
                <element name="symbol" type="string" />
            </all>
        </complexType>
    </element>
</schema>

<message name="GetStockPriceInput">
    <part name="body" element="PriceRequest" />
</message>
```

WSDL Interface Documents

As a whole, a WSDL interface document will consist of multiple port types, operations and messages. Together they define the interface of a service, separate from any deployed instance of the service. The interface can be **shared by many interchangeable services**.

```xml
<definitions xmlns="http://schemas.xmlsoap.org/wsdl"
    name="StockQuote"
    targetNamespace="http://example.com/stockquote">

    ...

</definitions>
```

Implementation WSDL

WSDL is also used to give details on how to use an abstract interface with a given service. Implementation details include the URL of the service's web server and the underlying protocol to use (typically HTTP).

While both the abstract definition and specific implementation details can be one file, they are often **split into two file**, so that an abstract interface can be imported and re-used by many implementation documents.

Bindings

A binding describes a concrete binding of a **port type** component (and its operations) to a **particular concrete message format and transmission protocol**. For example, one may specify the use of SOAP over HTTP, another may specify some other means.

Within a binding, further transport and encoding information is provided for each message of each operation of the port type.

```
<binding name=""”StockQuoteSoapBinding type="tns:“StockQuotePortType>

    <wsoap:binding style“=”document
        transport="http://schemas.xmlsoap.org/soap/http" />

    <operation name="GetLastTradePrice">
        <wsoap:operation soapAction="http://example.com/GetLastTradePrice" />
        <input>
            <wsoap:body use="literal" />
        </input>
        <output>
            <wsoap:body use="literal" />
        </output>
    </operation>
</binding>
```

Ports

A port of a web service is a similar idea to the TCP ports of a host (*see more: Transmission Control Protocol (TCP), page 26*). A port is **one channel of communication** to which messages of a particular purpose and format can be sent to and received from. Each port has its **own URL and binding**.

Clients send messages to that URL, conforming to the message schemas of the **binding's port type** and to the **binding's transport and encoding details**.

Services

A WSDL service is a **collection of ports**. They tie together all of the other parts of the interface into **one, named, whole definition of a web service**.

```
<service name="StockQuoteService">
    <port name="StockQuotePort" binding“=tns:StockQuoteBinding">
        <soap:address location="http://example.com/sq"/>
    </port>
</service>
```

Universal Description, Discovery and Integration (UDDI)

Web services cannot be used if they cannot be found, and the aim has always been for services to be widely re-used. UDDI is a **directory service specification** that has been taken as a de-facto standard for discovering web services.

UDDI is itself a web service: it has WSDL-defined port types for publishing descriptions of services and for discovering services.

A service description can contain information on owners of the service, the functions it performs, its WSDL interface, etc.

The Semantic Web

The web contains a wealth of information, but it is not always written in a way that is **easy for software to parse and use**. If software could search for and use the information on the web it could potentially be a lot more useful.

The idea of semantic web is to **include computer-readable information** on the web, alongside the current human-readable information.

Resource Description Framework (RDF)

Several technologies are required to make the semantic web work, the first of which is a **data structure in which to make the computer-readable statement**. The structure used is **RDF**.

Statements

An RDF document is a set of **statements**, which asserts something about a resource (often, its relation to another resource). Every statement contains three parts:

- The **subject**: which resource the statement is about (e.g. a URL to these notes).
- The **object**: what resource or value the statement is about (e.g. 'Mark Ormesher').
- The **predicate**: how the subject and object are related (e.g. 'creator').

Statements are written as `subject predicate object ..`

Resources

The **subjects** (and sometimes **objects**) of RDF statements are resources, which are things that are identifiable by a URI (such as a web page, a book, an abstract concept, etc.).

URIs are also used to give **identifiers for predicates**. URIs are written between `<...>` brackets in a statement:

```
<http://markormesher.co.uk/cs—notes>           <!—— subject ——>
    <http://purl.org/dc/terms/creator>         <!—— predicate ——>
        <http://markormesher.co.uk/profile> .  <!—— object ——>
```

Vocabularies

A vocabulary is a **set of terms defined to allow descriptions** in some particular domain (similar to namespaces). Each term is a **URI**, and all URIs in a vocabulary start with the **same prefix**.

For example, the vocabulary `http://purl.org/dc/terms` describes the creators and publishers of documents and other library-related data, such as:

- `.../creator`: links a subject to its creator.
- `.../publisher`: links a subject to its publisher.
- `.../isReplacedBy`: links a subject to a newer version of it.

Prefixes and Turtle

RDF statements can be encoded in different formats, including XML. We will use a formal called **Turtle**.

Because they are long we can **abbreviate URIs to prefixes**, where the prefix replaces the vocabulary URI. For example:

```
#prefix mo: <http://markormesher.co.uk/> .
#prefix dc: <http://purl.org/dc/terms/> .

mo:cs—notes dc:creator mo:profile .
```

Values

The **objects** of RDF statements can also be values (strings, integers, etc.). For example,

```
#prefix mo: <http://markormesher.co.uk/> .
#prefix foaf: <http://xmlns.com/foaf/0.1/> .
#prefix dc: <http://purl.org/dc/terms/> .

mo:profile foaf:firstName "Mark" .     <!—— value object ——>
mo:profile foaf:lastName "Ormesher" .  <!—— value object ——>
mo:cs—notes dc:creator mo:profile .    <!—— resource object ——>
```

RDF Graphs

A **set of RDF statements** is often called an RDF graph, because the information forms a graph with **subjects and objects as nodes** and **predicates as labels**.

Web Ontology Language (OWL)

Ontologies

RDF allows us to make computer-readable statements about resources, but it **does not (by itself) allow software to reason about the statements** to determine how to apply the information.

To allow this, we need to encode something about the meaning of resources, such as what kind of thing a resource is (a person, a book, etc.) and what is known about resources of that type (name, author, etc.).

Data encodes this meaning is called an **ontology**.

OWL

OWL is a language for **encoding ontologies** in RDF. An OWL ontology defines a **vocabulary of terms** but also says **how those terms relate to each other**, in order to give extra meaning for software to reason over using components called **reasoners**.

Classes and Individuals

The first kind of statements OWL allows us to make is to say what **class** a resource belongs to, using the predicate `rdf:Type`. For example, the following statement asserts that I am a kind of person (i.e. an instance of the class `Person`).

```
mo:profile rdf:Type ex:Person .
```

The prefixed URI `ex:Person` is a term in our ontology representing the class of all people.

`rdf:Type` is so common that it can be abbreviated to **`a`**:

```
mo:profile a ex:Person .
```

In the statement above, `mo:profile` is said to be an **individual**, because it is a **specific thing** in the world.

Multiple Classes

A resource can be of multiple classes, which can be expressed as follows:

```
mo:profile a ex:Person ;
           a ex:Man ;
           a ex:Student ;
           a ex:Developer .
```

Class Hierarchies

We would not want to have to say that every `ex:Man` is also an `ex:Person`, so we can declare that one is a **subclass** of the other:

```
ex:Man rdfs:subclassOf ex:Person .
ex:Woman rdfs:subclassOf ex:Person .
```

Now, `mo:profile a ex:Man .` also implies `... a ex:Person .` as well.

Properties and Data Types

Consider an RDF statement about OWL individuals:

```
mo:profile ex:worksIn ex:London .
```

In OWL, the predicate `ex:worksIn` is called a **property**. We can say more about a property's meaning using OWL by stating its **domain** (things it describe) and its **range** (values it can take):

```
ex:worksIn rdfs:domain ex:Person ;
           rdfs:range ex:City .
```

A similar notation can also specify the data type of value objects, borrowing the types from XML schema (*see more: XML Schema, page 46*).

```
ex:hasName rdfs:range xs:string .
ex:hasAge rdfs:range xs:integer .
```

Social Methodology

Getting people to agree on an ontology[1] is **difficult**. The more people who need to agree, the harder it becomes. If ontologies are imposed, people will disagree and choose not to use them. Instead, a **social approach** is used:

- Let **small groups** agree on **small ontologies**.
- Use **mappings** to combine them into **larger ontologies**.

Ontology Mappings

OWL provides vocabulary to map between two ontologies.

We can say that one class is equivalent to another, so any instance of one is assumed to be an instance of the other:

```
my:Person owl:equivalentClass your:Human .
```

Similarly, we can say that one individual is the same as another, even if they use different URIs:

```
mo:profile owl:sameAs github:markormesher .
```

[1]or anything, really

SPARQL

SPARQL is an SQL-like **query language** used to extract knowledge from stores of RDF data (**triple stores**. As it is designed for use on the web, there is also a **SPARQL Protocol** for sending queries to online triple stores and returning the results.

SPARQL Queries

A basic SPARQL query **finds all of the statements** (or combinations of statements) that follow a particular pattern, and **returns some subjects and/or objects** of those statements.

For example, we may want to:

- Retrieve the email address (the object of statements with the `foaf:mbox` predicate)...
- ...and the first name (the object of statements with the `foaf:firstName` predicate)...
- ...of everyone I know (statements following `mo:profile foaf:knows ? .`).

Pattern Variables

We need variables to represent parts of the data we are looking for, and we use `?var` or `$var` to represent these. The query described above could be represented as:

```
{ mo:profile foaf:knows ?x .
  ?x foaf:firstName ?fName .
  ?x foaf:mbox ?mbox }
```

Example Query

The pattern variables above can be used when expressing the query in full:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?mbox ?fName
WHERE
    { mo:profile foaf:knows ?x .
      ?x foaf:firstName ?fName .
      ?x foaf:mbox ?mbox }
```

This query returns only `?mbox` and `?fName`, because we don't care about the other variable.

Query Results

Results can be represented as a table, where column headings will be variable names and each row will contain a set of bindings to these variables:

?mbox	?fName
alan@example.com	Alan
steve@example.com	Steve

Semantic Web Pages

RDF can be stored in triple stores, but the original intention of semantic web was to provide machine-readable knowledge **alongside** human-readable web pages. This means that **RDF needs to be embedded into HTML**.

This RDF data is not presented to the user, but can be extracted by software to provide additional information about the content and resources.

RDFa

RDFs allows **RDF to be embedded into HTML**.

If we add a `property` attribute to an element making up test, the attribute value is a **predicate** relating the web page to the text.

For example, if the page at `http://markormesher.co.uk/cs—notes` contains...

```
<h2 property="http://purl.org/dc/terms/title">CS Notes</h2>
```

...then the following RDF statement is embedded into the page:

```
<http://markormesher.co.uk/cs—notes>
    <http://purl.org/dc/terms/title>
        "CS Notes" .
```

By default, the **subject** of all embedded RDF statements is the web page itself. We can also embed arbitrary RFD, with any subject, using the `resource` we are making statements about within a given HTML element:

```
<div resource="http://markormesher.co.uk/blog/post1">
    <p property="http://purl.org/dc/terms/title">My First Blog Post</p>
    <p property="http://purl.org/dc/terms/created">2016—11—17</p>
</div>

<div resource="http://markormesher.co.uk/blog/post2">
    <p property="http://purl.org/dc/terms/title">My Second Blog Post</p>
    <p property="http://purl.org/dc/terms/created">2016—11—18</p>
</div>
```

DBpedia

One of the largest semantic web projects is DBpedia, an open collaboration to create a machine-readable translation of Wikipedia. Using DBpedia RDF statements, software should have access to all of the same information that Wikipedia offers to humans.

The RDF statements currently describe over 20 million subjects and use an ontology with over 350 classes, although some repetition exists for different languages.