Unit 1. Introduction

Outcomes Summary. You will learn

- to define a distributed system (DS) and describe and justify the characteristics of all DSs
- to classify DSs by various important aspects, appreciate the challenges in accurately specifying the behaviour of a DS
- about some common ways to distribute responsibility among networked nodes
- about network abstractions avialable to DS developers
- about basic tools for deploying and managing DSs

Further Reading: CDK2005 1, 2.3, 3, 4.

1.1 Goals of this module

1.1.1 Motivation

A single computer is a powerful tool but a bunch of computers working together raise their shared potential to a new level.

A *distributed system* (DS) is a software deployed on a networked set of computers (nodes) that makes these computers cooperate towards a shared goal exclusively through exchanging messages with each other.

DSs have been developed and studied ever since the first computer networks were built at academic institutions. One could argue that the history of DSs started before computers were built because other networked devices, such as phones and phone exchanges, formed systems that have a lot in common with networked computers.

All of us are likely to interact with a DS on a daily basis. Some of today's most commonly used DSs are:

- phone networks (nodes = phones and exchanges)
- email (email clients, SMTP servers and mail-box servers)
- World-Wide Web (web servers and browsers)
- Internet search engines (high-performance clusters)
- enterprise systems (database servers and clients)
- file sharing systems such as BitTorrent (upload+download peers)
- electronic stability-enhancing system in a car (conrollers for individual brakes, differentials, engine injection, etc.)
- movie-effect renderers (CPUs in high-performance clusters)

Note that the Internet is a massive DS with a very general purpose, ie to host more specific DSs such as the ones above.

Given such a variety of DSs playing important roles, it is a subject that a computer scientist cannot ignore. Distributed systems understanding and distributed programming skills are an import asset and it is expected that its value is going to grow for some time to come.

1.1.2 Aims

DSs is a vast and deep subject and we can attempt to address only a small subset of fundamental architectures, technologies, techniques and algorithms to any substantial depth. There is probably a good amount of new concepts to grasp and some challenging programming scenarios to explore. All these should help you when extending an existing DS or designing a new one to eg

- make competent decisions about distributed architectures and tools;
- know where to start looking for help;
- quickly understand manuals and articles.

Moreover, you will be armed with a working knowledge of some of the distributed programming technologies that are used in industry and are expected to be used more and more.

1.1.3 Assessed Outcomes

It is very desirable that CS3250 helps you achieve the above aims. Nevertheless, these achievements are long-term and hard to measure. You will demonstrate sufficient understanding of DSs by being able to:

- name successful/typical DS applications,
  explain their main features
- describe main DS architectures and frameworks,
  main differences between them
- analyse DS requirements and design a suitable solution
- read and understand simple DS written in Erlang
- develop simple DS using
  - Java RMI
  - Web Services (RESTful, stateless, stateful) in Java

You are expected to attain these skills by practice in practicals and class exercises, with the help of these notes and other sources of conceptual information. It is also highly recommended that you help each other while learning, for which you can also use the BlackBoard on-line forum. Additionally, you are welcome to arrange a meeting with the module tutor or visit the tutor at the office hours.

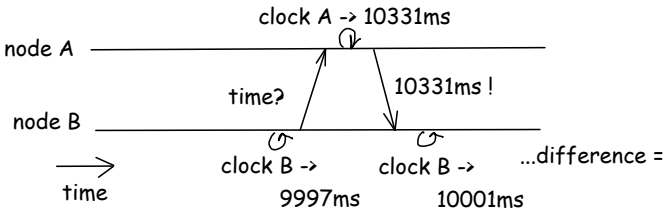


Figure 1.1: The Cristian protocol to communicate time on each other's clocks.

1.2 Characteristics of DSs

Our definition states that nodes in a DS do not share resources except that they can send each other messages via a network. From this we can deduce the following other important properties shared by all DSs.

1.2.1 Concurrency

In a DS we cannot avoid concurrency as multiple nodes operate independently in parallel. When nodes communicate with other nodes, the issues of synchronisation encountered in multi-threaded programming apply here as well. For example, a phone exchange needs to deal with several concurrent calls to the same phone, usually by giving exclusive access to one of them. This example also illustrates that nodes in distributed system are often multi-threaded themselves to better deal with concurrently incoming messages and multiple responsiblities.

1.2.2 No global clock

It is impossible for two nodes on a network to synchronise their clocks perfectly. If they agree to make an exchange of several messages in a quick succession including information about the time in their internal clocks, they can work out an estimate of how long the messages took to transfer and from that an estimate of the difference between their clocks (see Figure 1.1). Nevertheless, there is no way nodes can work out exactly what the difference between their clocks are because neither of them can measure how long a single message between them takes.

The most accurate way to try and synchronise the internal clocks of different computers is by receiving time signals from the same radio or satellite source. Nevertheless, even these sources suffer from small inaccuracies and thus do not offer perfect synchronisation for DSs.

The lack of a global clock is usually not a big problem but designers and developers of DSs have to remember this fact and avoid certain techniques that rely too much on time-stamps.

To give a simple example of inappropriate use of time in a distributed environment, consider a system that first uses a simple but potentially inefficient method to solve a problem and when it fails to solve it within a time limit, it resets itself and attempts

another method to solve the problem. If multiple nodes independently detect the time-out, it could happen that some node discovers a solution just as some of their collaborators give up and thus cannot process the solution. Such global decisions have to be carefully coordinated among the nodes without relying on clock agreement.

1.2.3 Independent failures

A failure in a non-distributed system is usually communicated to all affected threads as soon as possible (unless the failure caused the OS or hardware to stop or reboot). In DSs this is often not possible as a DS can fail in new ways.

For example, a network failure is indistiguishable from a very slow network and thus hard to detect. It leaves all system nodes running for some time unaware of the problem. When the nodes become aware of the problem, they could be left without any means of communicating with each other to coordinate a recovery or shutdown.

Similarly, when a node in a DS dies, the others are left running and it may take them some time to discover that.

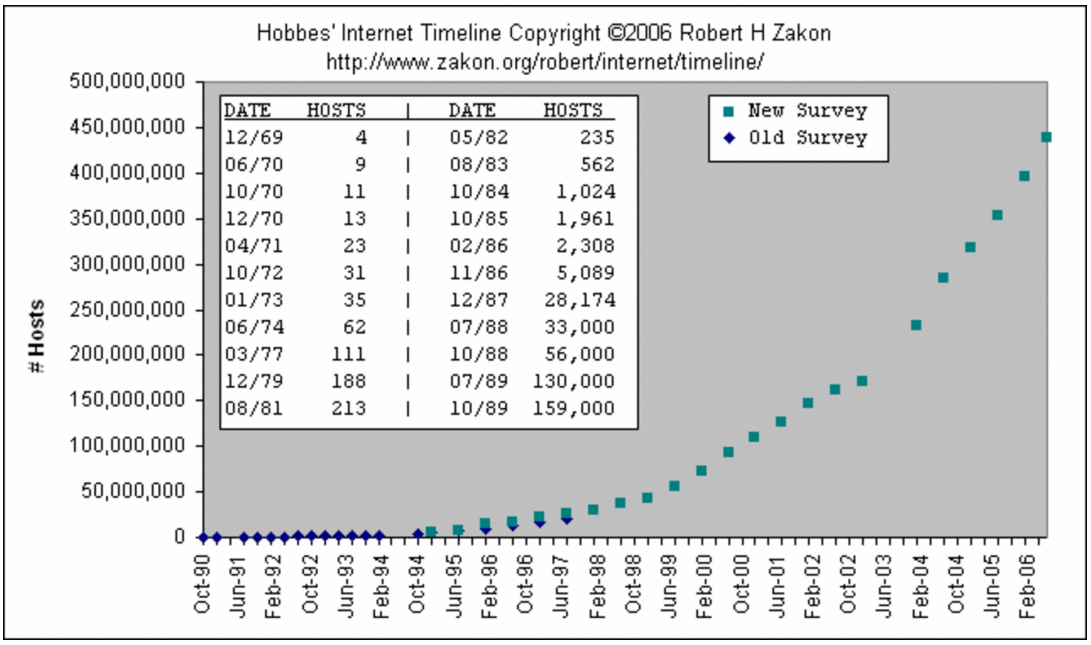
1.3 Classification of DSs

Here we discuss how DSs differ in terms of properties visible to users, administrators and developers of the system. We start with properties visible to users and progress towards properties relevant only to developers.

1.3.1 Purpose

The example DSs in Section 1.1.1 represent a wide spectrum of different distributed architectures and technologies as well as different underlying reasons for choosing a distributed solution. Inspired by these examples, let us classify the fundamental reasons why we need distributed computing:

- *information transfer* between distant or moving users and/or controllers
  (most common reason; eg phones, email, car sub-systems)
- *resource sharing*
  (eg sharing printers & disks, BitTorrent, WWW)
- *performance enhacement*:
  - *data too large* to fit on one computer's memory
    (eg search engines)
  - *task too complex* to be solved using a single computer in a reasonable time
    (eg movie renderers, search engines)
- *protecting against failures* of individual computers by duplicating and/or dispersing responsiblity among computers
  (eg peer-to-peer file sharing, Internet packet routing)



DATE	HOSTS	DATE	HOSTS
12/69	4	05/82	235
06/70	9	08/83	562
10/70	11	10/84	1,024
12/70	13	10/85	1,961
04/71	23	02/86	2,308
10/72	31	11/86	5,089
01/73	35	12/87	28,174
06/74	62	07/88	33,000
03/77	111	10/88	56,000
12/79	188	07/89	130,000
08/81	213	10/89	159,000

Figure 1.2: The growth of Internet hosts

1.3.2 Scalability

Scalability is a measure of the ease with which a DS copes with an increasing load. The load could be measured by the amount of nodes, requests, resources, users, etc. or whatever else is appropriate.

A perfectly scalable system would manage an arbitrary imaginable load and at the same time:

- not require any modification of its software;
- the cost of additional hardware would be proportional to the extra load (ie cost of hardware is $O(n)$ where n is the load);
- the performance of basic internal operations (such as resolving a node name or sending a message) would decrease logarithmically with the load (ie time taken for such operations is $O(\log(n))$).

It could be argued that the Internet is close to being a perfectly scalable DS. Its size has grown beyond anyone's expectations (see Figure 1.2) and yet, the basic underlying TCP/IP software from around 20 years ago still works on today's Internet. Despite its growth the Internet is not getting slower (arguably also helped by the increase in the power of the nodes and improving network links).

A typical Web resource is harder to scale because it usually resides on a single server with a limited network bandwith and processing power. Although there are ways to make it scale (as shown by Google for example), it requires replication and load balancing — tricks that are not part of the original Web protocols. Naturally, a Web server is the *bottleneck* of the Web service.

A significant example of a bottleneck destroying scalability is in the predecessor of the Domain Name System (DNS). (DNS is a DS on the Internet that translates internet computer names such as `duck.aston.ac.uk` to numerical IP addresses.) When the Internet consisted of a few hundred nodes, there was a single master table listing all the nodes and their IP addresses. Whenever another computer needed to lookup a name, it would download this file. This technique was clearly not scalable because the table would need to be updated and downloaded very often with half a billion computers in the network. Moreover, the table would have to be over 30GB large in today's Internet. Current DNS is scalable because most DNS servers have full knowledge only about local nodes and pass resolution of remote nodes to higher-level DNS servers, who delegate it to appropriate local DNS servers.

1.3.3 Security

A DS is fully secure if there is no way anyone can successfully attack it. An attack tries to either cause the system to do something unintended or prevent it from doing what it is supposed to be doing. The unintended behaviour caused by an attack is typically:

- giving unauthorised access to data;
- unauthorised alteration of data; or
- a complete break-down of the system.

It is generally believed that it is impossible to make a DS fully secure. Nevertheless, security levels can vary a lot between DSs.

Some DSs rely solely on a firewall to prevent any access from outside the intranet they are confined to. They are not secured against attacks from inside the intranet.

Other DSs are secured against attacks that interfere with their messages but not against attacks coming from inside its own nodes. This is the most common case. For example WWW services using the https protocol use encryption to:

- verify the identity of the web server (and sometimes also the browser);
- make it almost impossible for third party to understand the messages or modify the content of messages.

Nevertheless, if someone gains access to the computer on which the server is running, they could potentially read and modify sensitive data, stop the server, etc.

A hard one to avoid is the so-called *denial-of-service* (DOS) attack in which one or more important nodes are overloaded by meaningless messages to the degree that they become unresponsive.

Systems using so called *mobile code* are open to other attacks that are difficult to defend against. In these systems, programs are sometimes sent within messages with the intention that the programs should be executed on the remote node (eg Javascript executed in Web browsers). How can the receiver be sure that the program will do what it says "on the tin" (eg not send your password unencrypted over the network)? How can the sender be sure that the receiver will not work out from the program more than it should know about the internal working of the sender and attack the sender (eg send many AJAX calls to overload a web server)?

1.3.4 Heterogeneity support

A DS can support heterogeneity when it can be easily deployed using a mixture of various infrastructures. Such variations can apply in particular to:

- networks
- computer hardware
- operating systems

It is usually desirable that DSs support as much heterogeneity as possible. Nevertheless, these differences are costly to bridge, and thus most DSs can be deployed only on fairly specific subset of available infrastructures.

Network types. The Internet has helped to unify most of the World's network types. It is based on open standards that provide an abstract view of any sub-network as a set of nodes with physical connections between some of them, ignoring any differences in physical connectivity (such as cables or radio) and low-level data exchange protocols. The IP protocol provides logical links between nodes that do not have a physical link by routing messages between them through intermediate nodes. The TCP protocol makes these logical links more reliable by the ability to detect and sometimes correct failures.

Thus when using the Internet as an infrastructure for a DS, it automatically supports heterogenous networks.

Hardware, OS. Most modern programming languages do a good job at hiding the differences of hardware and OS. For example, programs written in Java and Erlang usually run on almost any combination of hardware and OS with the same effect.

On the other hand, writing programs in lower-level languages such as C/C++ that can be ported between different hardware and OS is much more difficult. For example, most C programs written on 32-bit Intel-386 Linux require some work to be usable also on a 64-bit Intel Linux, mainly due to different default sizes of the `int` type and memory pointers. Porting these programs to Microsoft Windows is even harder because they rely on many system libraries and kernel calls that would have to be simulated.

1.3.5 Openness

A DS is open if there is appropriate support for it to be extended and adapted by any programmer. Clearly, there are degrees of openness. A perfectly open system would:

- have all its protocols and APIs public and well-documented;
- provide well-documented access methods to all its resources;
- be well-designed for extensibility (eg uniform protocols for accessing similar resources, good cohesion and coupling);

- support heterogeneity of OSs and hardware;
- be extensible in any mainstream programming language.

The Internet and WWW are good examples of open DSs. They are based on well-documented protocols and standards (TCP/IP, DNS, HTTP, HTML, CSS, ...) that support full heterogeneity and are supported in almost any programming language. The standards make it clear how to access each node and its resources. Consequently, it is not hard to create a home network based on Internet standards, connect it to the Internet, deploy a web server and put new resources on it. More specifically, it is not too hard for a programmer to write their own correct TCP/IP stack or web server or, more probably, tailor existing ones to their needs.

Commercial DSs and DSs confined to a single organisation are often closed. Closed DSs can build on or extend open general purpose DS such as the Internet and WWW.

It is often believed that closed systems are more secure because attackers find it harder to work out how the system works. Experience shows that this is not always true because attackers can attack via well-known general-purpose system and networking libraries on which the closed system is built. On the other hand, an open system can be very secure because it tends to have been checked by thousands of competent programmers and thus have less bugs. (Note that an open system can hide its data using encryption — while its architecture and programs are open, its data can be easily closed.)

1.3.6 Transparency

The transparency of a DS is the measure to which it hides from its users (and/or developers) the fact that it is distributed. Depending on the underlying purpose of making the system distributed, it is more or less desirable that the system appears as a standalone application to the users.

Sometimes, only certain aspects of distribution should be hidden while others must be clearly visible to the user. For example, when accessing a Web search engine, we want to know which Web server address it resides on so that we are not cheated by a rogue look-alike "service" that is likely to send us to malicious sites. On the other hand, we do not need to know exactly which one of the many back-end servers is talking to our browser or which subset of the database servers was used to answer our last query.

An instant messaging application usually presents to its users a single world entity where anyone can log-in and chat to anyone else who has logged in. The networking behind this behaviour is transparent to the users.

On a more subtle level, even the effects of being distributed can be made transparent. The following are the most commonly recognised forms of transparency in distributed processing (based on Coulouris et al 2005, p23):

- *Network transparency*:
  - *Access transparency* enables local and remote resources to be accessed using identical operations.
    (Eg cannot tell which files are remote in a file chooser.)

  - *Location transparency* allows resources to be accessed without knowledge of their physical or network location.
    (Eg Internet domain names, consequently URLs.)

- *Mobility transparency* allows the movement of resources and nodes within a system without affecting the operation of users or programs.
  (Eg mobile phone networks, Internet with Mobile IP protocol.)
- *Failure transparency* allows the (automatic) concealment of hardware and software faults so that users do not even notice or at least can continue in their work as if it did not happen.
- *Performance transparency* allows the system to hide from its users the effects of variations in its load by reconfiguring itself so that it performs always to the same standard to all users.
- *Scaling transparency* is similar to performance transparency except that the system reacts to high load by expanding its size without any change to its structure or visible behaviour.
- *Replication transparency* enables the use of multiple copies (ie replicas) of a specific resource without any effect that users of that resource could observe.
- *Concurrency transparency* makes it possible to carry out certain tasks in parallel threads that do not interfere and thus have the same effect as if the actions were perfomed by a single thread in a series. Consequently, the user cannot tell that parallelism took place.

The recent rise of *cloud computing* reflects the desire to implement as much transparency as possible — cloud services are distributed systems often featuring network, failure, performance, scaling and replication transparency.

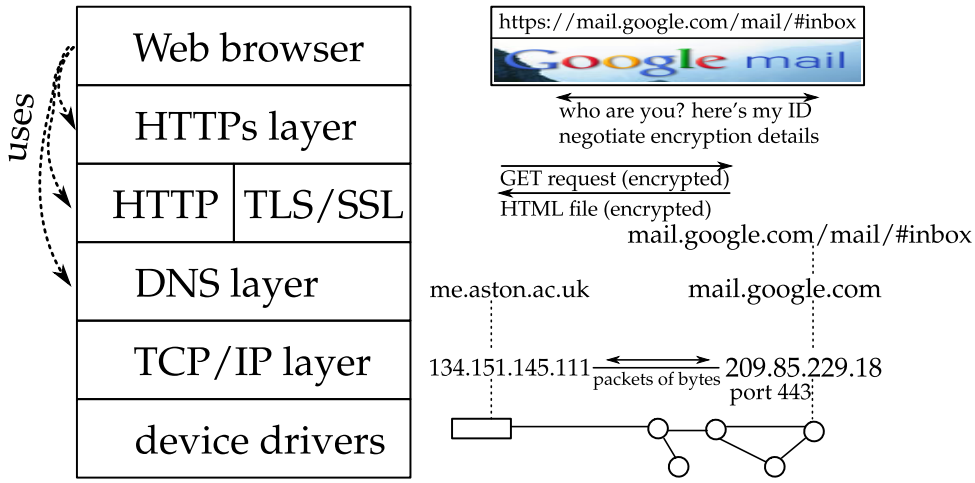
1.3.7 Architecture

Architecture of a DS determines how the responsiblity of the system is spread among the nodes and how these nodes synchronise with one another to fulfill the system's responsibility. The two most common extreme architectures are:

- *Tiered client-server architectures.* Nodes are divided into layers, the top layer serves end users and each layer uses services of the layer underneath it. In the simplest case there are only two layers: clients and servers.
- *Peer-to-peer (P2P) architectures.* All nodes have the same function, they only differ by their data and location.

We will study issues specific to these main architectures in the following two units.

Another important DS architecture is one where nodes are dedicated to processing certain type of data coming in and making its results available to other nodes who may take is as input and do further processing. Such architecture is sometimes called data-flow driven or data-driven. Among prominent examples are systems using the



MapReduce approach first introduced by Google. In a MapReduce architecture, some nodes divide a large dataset into smaller chunks, while other nodes do some processing on these chunks and yet others collate the results together. The system can thus process a very large dataset quickly.

1.4 Network abstraction

The most low-level classification of DS focuses on the type of network the developers are targetting the system to run upon. The most important aspect is that the developers typically do not need to know much about the physical wires and switches the network is built from nor the electric signal-level protocols that are used on the wires. Networking is utilised at a certain level of abstraction. Each level of abstraction available to developers is formed by a software library, and sometimes defined by a set of standards. Eg, the developers of a Web browser work with one or more libraries implementing the HTTP standard, which gives them a view of a network where nodes are files that can be fetched using simple GET requests together with MIME classification of the type of content such as HTML or JPG.

1.4.1 Stacking abstraction layers

In some sense, a library such as a HTTP one used by Web browser developers, is a DS, whose users are the developers of higher-level DSs. The developers of the HTTP library build it on top of a lower-level abstraction of the network, typically a library implementing the Internet TCP/IP and DNS protocols. The TCP/IP library is built on top of a set of network drivers that were built with understanding how to control individual ethernet adapters or other similar devices. We get a stack of layers with the most concrete and hardware-oriented ones at the bottom and higher-level ones oriented at end-system developers at the top (see Fig. 1.4.1). Each layer builds on one or more layers below it.

A typical DS developers will need to use only the top layer, reducing the coupling of their DS with the stack. Nevertheless, sometimes the developers need to tap directly into some lower layers, eg a Web browser uses HTTPS and HTTP as well as the DNS layers directly. The main reason for this is that protocols such as HTTP do not deal with node addressing but inherit and expose concepts, such as a host IP address or a DNS name, from the lower layers. In this module we will only consider DSs built on top of the Internet protocols and usually using only a layer above the Internet protocol that inherits some concepts from the Internet, such as DNS host names, but provides its own higher-level concepts, such as URLs and remote objects.

The top general-purpose layer in the networking stack is often one that presents the network using concepts native to a high-level programming language, eg nodes=objects and messages=method calls. This kind of layer is called a *middleware*. A middleware can be specific to a particular programming language, but in the interest of openness, it is preferable to use middleware that supports a wide range of programming languages. A large proportion of this module is dedicated to studying and learning to use various types of middleware.

As middleware typically inherits some concepts and properties from lower layers, its users cannot be completely unaware of the lower layers. We shall therefore first review basic concepts of the Internet and Web protocols that define the layers that often support a middleware.

1.4.2 Internet protocols

TCP/IP. We do not need to study the details of TCP/IP. For a DS designer and programmer, it is often sufficient to keep in mind that the TCP protocol allows any two computers on the Internet to establish any number of logical channels with each other through which they can send big or small messages or stream a long sequence of bytes in either direction.

To establish such a connection, one of the computers has to initiate it and the other one must be *listening* in order to accept the initiative. To make it easier for a computer to be listening to requests for TCP connections of many kinds (eg for different applications), a listener is always attached to a number called *port*. For example, a web server typically listens on port 80. If there are multiple web servers on the same computer, only one of them can listen on 80; the others have to listen on non-standard ports.

Most often, what we just said is all that a DS programmer needs to know about TCP because they use a middleware that hides TCP from them (perhaps except for port numbers). In some cases, however, one prefers a lower-level control of communication and uses TCP directly. We will not be doing it in this module.

DNS. At the TCP level, computers are addressed using their IP numbers. The Domain Name System (DNS) is used to give computers more memorable and better structured names.

For a DS programmer DNS is a service that they can use to translate fully qualified names such as external.aston.ac.uk to the corresponding numerical IP addresses should they need to know them. A few more facts are worth keeping in mind:

- DNS can map several names to the same IP (eg gnasher.aston.ac.uk = external.aston.ac.uk);
- one name can be mapped to multiple computers, so subsequent look-ups for that name give different IPs (eg www.google.co.uk);
- DNS look-up can be sensitive to location (eg www.google.com = www.google.co.uk in the UK).

1.4.3 Web protocols

Standards and protocols such as URL, HTTP, etc. are mostly used in the Web for transporting HTML and other documents to browsers. Nevertheless, they play an increasing role in DS programming thanks to the rise of the Web Services technologies (ie various middleware based on XML and Web protocols).

HTTP. HTTP is a so-called request-response protocol because a typical HTTP communication consists of a request followed by a response. (Beware: this is not the case for all protocols, some are much more complex than this.) Each HTTP request specifies a so-called *method* to indicate its purpose. There are 8 different methods, the most common ones being:

- GET for obtaining a remote resource
- POST for initiating some operation on a remote resource
- PUT for storing a local resource at a remote location
- DELETE for permanently removing a remote resource

Nowadays, the HTTP methods GET and POST dominate the Web and are sometimes used for all kind of actions, contrary to their original purpose stated above. In the unit on RESTful Web Services we will learn how general DS programming can be achieved while staying true to these principles.

URLs. Each HTTP request needs to provide an address of the computer at the other end and moreover specify what resource at that computer the request is about. DNS and TCP ports are used as a part of this address and are usually used to establish a TCP channel over which the HTTP communication is realised.

Putting together TCP, DNS identification and local resource name lead to the concept of URL. The name of the protocol was added to make URLs independent of HTTP to become the primary means of addressing resources on the Internet.

HTTPs. HTTPs is HTTP over a secure channel. The channel is secured using the TLS/SSL protocol for node authentication and message encryption. Figure 1.4.1 illustrates how this protocol is used for classical browser-server communication. It can be used analogously in Web Services with invisible clients rather than Web browsers.

This is a good example of one protocol being built from lower level protocols: HTTPs is built on top of TLS/SSL and HTTP. The way they are put together is illustrated in Figure 1.3.

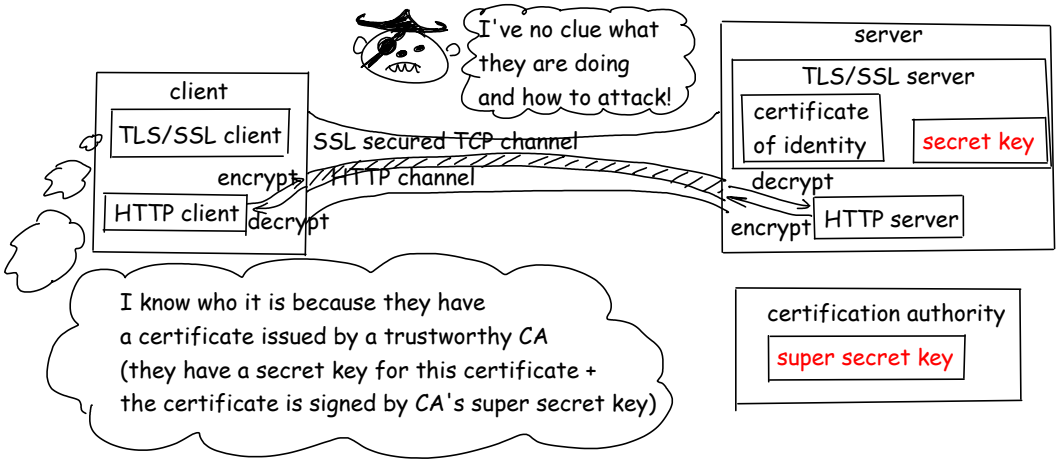


Figure 1.3: A logical secure channel for HTTPs

1.4.4 Middleware services

One of the main function of middleware is to give programmers a more convenient way of referring to DS nodes in their programs and programming the sending and receiving of messages. Ie, a middleware provides an abstract view of the network, usually through a number of protocols for communication building on top of TCP/IP and perhaps HTTP. Being aimed at the DS programmers, these protocols are provided via programming language libraries.

The high level of abstraction in a middleware network model is usually characterised by

- convenient addressing of *logical nodes*; and
- convenient messaging of *data of various types*.

Convenient addressing. Addressing sometimes uses convenient internal names, hiding the actual Internet locations of nodes. Addressing logical nodes rather than physical nodes brings such benefits as being able to migrate logical nodes around the network or host several logical nodes on one physical node. Eg, in object-oriented middleware some objects usually play the role of distributed nodes and they can be assigned global names.

Convenient messaging. While at the lower layers network communication is in terms of sequences of bytes, at the middleware level we appear to be able to send and receive messages that contain structured data, similar to the data we can store in the variables of our programming language, eg strings, numbers and objects or object references or even mobile code.

Besides being able to communicate high-level data, middleware often helps nodes synchronise with one another in the process of sending and receiving messages. The most common sending modes are:

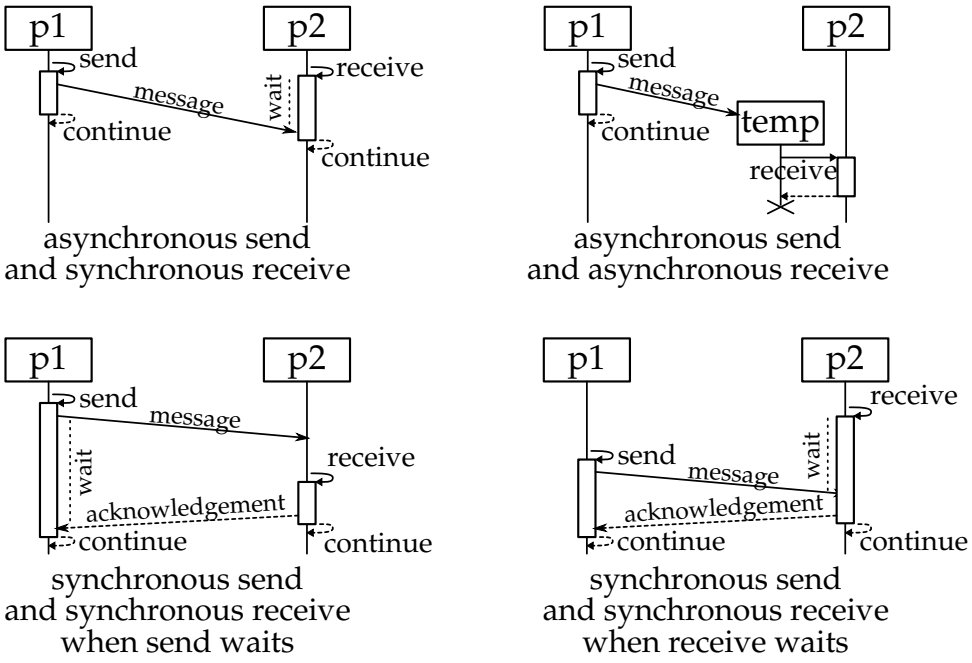


Figure 1.4: Various methods of sending a message between two nodes.

- *synchronous send to one node*: The sending thread waits until the message has been received (as with a phone call). As send and receive overlap in time, it is common to aggregate several messages in both directions into one session when both nodes communicate with each other. For example a remote method invocation involves a sending of paramter values to the callee and sending a return value to the caller.
- *asynchronous send to one node*: The sending thread does not wait, it may not even be able to find out whether the message has been delivered (as with an email).
- *asynchronous send to multiple nodes (aka multicast)*: Message is not addressed to a specific destination, it is instead addressed to a topic to which multiple nodes may be subscribed (as with Twitter).

Receiving also happens either synchronously or asynchronously. A synchronous receive is programmed by an explicit "receive" instruction for the executing thread, which will wait for a message unless one is waiting to be received already. A asynchronous receive executes a handler for the message in a thread that the programmer does not explicitly start. (It is helpful to imagine that the thread is started when the message arrive although it is not always the case.) The asynchronously receiving program has to anticipate the incoming messages and register a handler for them with the middleware so that the middleware knows which handler to execute when the message arrives.

Multicast Multicast is usually achieved by creating a virtual node address to which real nodes can subscribe. Whenever something is sent to the virtual node, it gets resent to all the subscribers. Such a virtual node is often called a *multicast group*. The efficiency of multicast depends on the support in underlying low-level protocols (usually

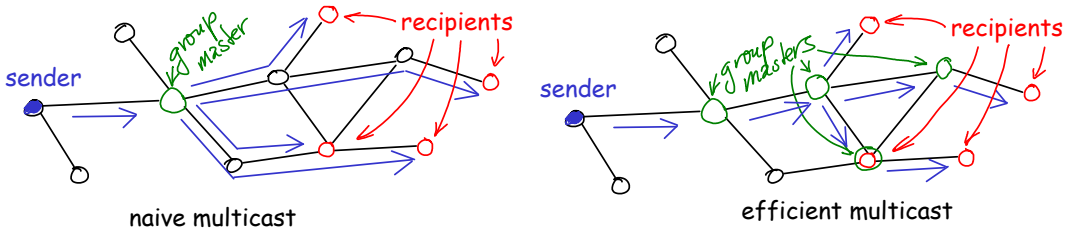


Figure 1.5: Naive and efficient multicast

the availabilty of IP-level multicast) and whether the chosen middleware is configured to make use of it.

In the absence of low-level efficient multicast, one can implement multicast at the level of middleware or directly in one's application. The easiest way to implement multicast is to model each group by a physical node that manages group membership and forwards messages to all group members one by one. (The way such a multicast group process works is very similar to the operation of a simple chat room server such as the one presented in the practicals and in unit 3.)

It is possible to implement a more efficient multicast facility by re-implementing the main idea of the low-level protocols, which is that the messages for a group should be forwarded not by one but by a hierarchy of special nodes. Such arrangement reduces the network load from $O(n \times m)$ to $(O(\log(n) \times m)$ where n is the size of the group and m is the size of the data. This is illustrated by the diagram in Figure 1.5.

1.5 Tools for managing distributed nodes

How do we start a program on remote computer? How do we fix a remote computer in a broken state but still reachable on the network? We shall briefly look at a selection of generic tools for these tasks. The selection is biased towards Unix style operating systems to complement the practical exercises.

Middleware often includes a remote management layer specialised at managing only those activities that nodes take through this particular middleware. We will not explore any middleware to this level in this module.

1.5.1 Remote shell

The simplest method of controlling another computer is to start a remote computer's command-line shell in a terminal and through this shell start programs on the other computer.

Advantages:

- works even for computers to which we have only a slow network link;
- most shell remote admin activities can be scripted and automated easily;

- can be very productive for some tasks
  (eg sophisticated regular renaming of lots of files in many directories);
- some programs are designed for console usage and have no good graphical interface.

Disadvantages:

- only suitable for console-based programs, which are harder to master;
- less productive for some tasks
  (eg searching among many files with no regular structure or clue).

The secure shell (SSH) is the most commonly used implementation of this idea. It is similar to HTTPs in the way it wraps a terminal emulation channel in a sophisticated virtual secure TCP channel. A single SSH connection can be configured to enclose multiple TCP channels and make remote ports appear as if they were local ports.

1.5.2 X-Windows thin clients

When using a workstation with an X-Windows a server it is possible to execute X-Windows programs on remote computers whose windows are controlled by the local workstation. Mouse and keyboard events that influence the remote application's windows are sent immediately over to the controlling program. The program controls the appearance of the windows by a selection of graphical commands and by creating and manipulating logical resources stored on the workstation such as images and typeset text. Thus the workstation can make use of graphical acceleration while giving all application control to a remote computer.

All communication between the workstation and the remote server is automatically and securely tunnelled over SSH when the program is started using SSH.

1.5.3 Desktop sharing

Another approach is to embed the whole desktop environment of a remote computer into a local desktop and thus gain complete control of the remote computer's desktop. This is frequently used by technical support teams to fix and configure customer's computers over a distance.

For Microsoft Windows computers this is the most convenient way of gaining remote access. The Remote Desktop protocol is an implementation that originated on Microsoft Windows but there are easily accessible implementations for all operating systems allowing desktop sharing in heterogeneous networks.

Desktop sharing requires more bandwidth than X-Windows clients and works best on intranets.