
THE CHALLENGES IN DISTRIBUTED SYSTEM:

Heterogeneity

The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks. Heterogeneity (that is, variety and difference) applies to all of the following:

- networks;
- computer hardware;
- operating systems;
- programming languages;
- implementations by different developers

Although the Internet consists of many different sorts of network, their differences are masked by the fact that all of the computers attached to them use the Internet protocols to communicate with one another. For example, a computer attached to an Ethernet has an implementation of the Internet protocols over the Ethernet, whereas a computer on a different sort of network will need an implementation of the Internet protocols for that network.

Data types such as integers may be represented in different ways on different sorts of hardware – for example, there are two alternatives for the byte ordering of integers. These differences in representation must be dealt with if messages are to be exchanged between programs running on different hardware. Although the operating systems of all computers on the Internet need to include an implementation of the Internet protocols, they do not necessarily all provide the same application programming interface to these protocols. For example, the calls for exchanging messages in UNIX are different from the calls in Windows.

Different programming languages use different representations for characters and data structures such as arrays and records. These differences must be addressed if programs written in different languages are to be able to communicate with one another. Programs written by different developers cannot communicate with one another

unless they use common standards, for example, for network communication and the representation of primitive data items and data structures in messages. For this to happen, standards need to be agreed and adopted – as have the Internet protocols.

Middleware • The term *middleware* applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages. The Common Object Request Broker (CORBA), is an example. Some middleware, such as Java Remote Method Invocation (RMI), supports only a single programming language. Most middleware is implemented over the Internet protocols, which themselves mask the differences of the underlying networks, but all middleware deals with the differences in operating systems and hardware.

Heterogeneity and mobile code • The term *mobile code* is used to refer to program code that can be transferred from one computer to another and run at the destination – Java applets are an example. Code suitable for running on one computer is not necessarily suitable for running on another because executable programs are normally specific both to the instruction set and to the host operating system.

The *virtual machine* approach provides a way of making code executable on a variety of host computers: the compiler for a particular language generates code for a virtual machine instead of a particular hardware order code. For example, the Java compiler produces code for a Java virtual machine, which executes it by interpretation.

The Java virtual machine needs to be implemented once for each type of computer to enable Java programs to run.

Today, the most commonly used form of mobile code is the inclusion Javascript programs in some web pages loaded into client browsers.

Openness

The openness of a computer system is the characteristic that determines whether the system can be extended and reimplemented in various ways. The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs.

Openness cannot be achieved unless the specification and documentation of the key software interfaces of the components of a system are made available to software developers. In a word, the key interfaces are *published*. This process is akin to the standardization of interfaces, but it often bypasses official standardization procedures,

which are usually cumbersome and slow-moving. However, the publication of interfaces is only the starting point for adding and extending services in a distributed system. The challenge to designers is to tackle the complexity of distributed systems consisting of many components engineered by different people. The designers of the Internet protocols introduced a series of documents called ‘Requests For Comments’, or RFCs, each of which is known by a number. The specifications of the Internet communication protocols were published in this series in the early 1980s, followed by specifications for applications that run over them, such as file transfer, email and telnet by the mid-1980s.

Systems that are designed to support resource sharing in this way are termed *open distributed systems* to emphasize the fact that they are extensible. They may be extended at the hardware level by the addition of computers to the network and at the software level by the introduction of new services and the reimplementation of old ones, enabling

application programs to share resources.

To summarize:

Open systems are characterized by the fact that their key interfaces are published.

Open distributed systems are based on the provision of a uniform communication mechanism and published interfaces for access to shared resources.

Open distributed systems can be constructed from heterogeneous hardware and software, possibly from different vendors. But the conformance of each component to the published standard must be carefully tested and verified if the system is to work correctly.

Security

Many of the information resources that are made available and maintained in distributed systems have a high intrinsic value to their users. Their security is therefore of considerable importance. Security for information resources has three components: confidentiality (protection against disclosure to unauthorized individuals), integrity (protection against alteration or corruption), and availability (protection against interference with the means to access the resources).

In a distributed system, clients send requests to access data managed by servers, which involves sending information in messages over a network. For example:

A doctor might request access to hospital patient data or send additions to that data.

In electronic commerce and banking, users send their credit card numbers across the Internet.

In both examples, the challenge is to send sensitive information in a message over a network in a secure manner. But security is not just a matter of concealing the contents of messages – it also involves knowing for sure the identity of the user or other agent on whose behalf a message was sent.

However, the following two security challenges have not yet been fully met:

Denial of service attacks: Another security problem is that a user may wish to disrupt a service for some reason. This can be achieved by bombarding the service with such a large number of pointless requests that the serious users are unable to use it. This is called a *denial of service* attack. There have been several denial of service attacks on well-known web services. Currently such attacks are countered by attempting to catch and punish the perpetrators after the event, but that is not a general solution to the problem.

Security of mobile code: Mobile code needs to be handled with care. Consider someone who receives an executable program as an electronic mail attachment: the possible effects of running the program are unpredictable; for example, it may seem to display an interesting picture but in reality it may access local resources, or perhaps be part of a denial of service attack.

Scalability

Distributed systems operate effectively and efficiently at many different scales, ranging from a small intranet to the Internet. A system is described as *scalable* if it will remain effective when there is a significant increase in the number of resources and the number of users. The number of computers and servers in the Internet has increased dramatically. Figure 1.6 shows the increasing number of computers and web servers during the 12-year history of the Web up to 2005 [zakon.org]. It is interesting to note the significant growth in both computers and web servers in this period, but also that the relative percentage is flattening out – a trend that is explained by the growth of fixed and mobile personal computing. One web server may also increasingly be hosted on multiple computers.

The design of scalable distributed systems presents the following challenges:

Controlling the cost of physical resources: As the demand for a resource grows, it should be possible to extend the system, at reasonable cost, to meet it. For example, the frequency with which files are accessed in an intranet is likely to grow as the number of users and computers increases. It must be possible to add server computers to avoid the performance bottleneck that would arise if a single file server had to handle all file access requests. In general, for a system

with n users to be scalable, the quantity of physical resources required to support them should be at most $O(n)$ – that is, proportional to n . For example, if a single file server can support 20 users, then two such servers should be able to support 40 users.

Controlling the performance loss: Consider the management of a set of data whose size is proportional to the number of users or resources in the system – for example, the table with the correspondence between the domain names of computers and their Internet addresses held by the Domain Name System, which is used mainly to look

up DNS names such as www.amazon.com. Algorithms that use hierarchic structures scale better than those that use linear structures. But even with hierarchic structures an increase in size will result in some loss in performance: the time taken to access hierarchically structured data is $O(\log n)$, where n is the size of the set of data. For a

system to be scalable, the maximum performance loss should be no worse than this.

Preventing software resources running out: An example of lack of scalability is shown by the numbers used as Internet (IP) addresses (computer addresses in the Internet). In the late 1970s, it was decided to use 32 bits for this purpose, but as will be explained in Chapter 3, the supply of available Internet addresses is running out. For this reason, a new version of the protocol with 128-bit Internet addresses is being adopted, and this will require modifications to many software components.

Growth of the Internet (computers and web servers)

<i>Date</i>	<i>Computers</i>	<i>Web servers</i>	<i>Percentage</i>
1993, July	1,776,000	130	0.008
1995, July	6,642,000	23,500	0.4
1997, July	19,540,000	1,203,096	6
1999, July	56,218,000	6,598,697	12
2001, July	125,888,197	31,299,592	25
2003, July	~200,000,000	42,298,371	21
2005, July	353,284,187	67,571,581	19

Avoiding performance bottlenecks: In general, algorithms should be decentralized to avoid having performance bottlenecks. We illustrate this point with reference to the predecessor of the Domain Name System, in which the name table was kept in a single master file that could be downloaded to any computers that needed it. That was

fine when there were only a few hundred computers in the Internet, but it soon became a serious performance and administrative bottleneck.

Failure handling

Computer systems sometimes fail. When faults occur in hardware or software, programs may produce incorrect results or may stop before they have completed the intended computation. Failures in a distributed system are partial – that is, some components fail while others continue to function. Therefore the handling of failures is particularly difficult.

Detecting failures: Some failures can be detected. For example, checksums can be used to detect corrupted data in a message or a file. It is difficult or even impossible to detect some other failures, such as a remote crashed server in the Internet. The challenge is to manage in the presence of failures that cannot be detected but may be suspected.

Masking failures: Some failures that have been detected can be hidden or made less severe. Two examples of hiding failures:

Messages can be retransmitted when they fail to arrive.

File data can be written to a pair of disks so that if one is corrupted, the other may still be correct.

Tolerating failures: Most of the services in the Internet do exhibit failures – it would not be practical for them to attempt to detect and hide all of the failures that might occur in such a large network with so many components. Their clients can be designed to tolerate failures, which generally involves the users tolerating them as well. For example, when a web browser cannot contact a web server, it does not make the user wait for ever while it keeps on trying – it informs the user about the problem, leaving them free to try again later. Services that tolerate failures are discussed in the paragraph on redundancy below.

Recovery from failures: Recovery involves the design of software so that the state of permanent data can be recovered or ‘rolled back’ after a server has crashed. In general, the computations performed by some programs will be incomplete when a fault occurs, and the permanent data that they update (files and other material stored

in permanent storage) may not be in a consistent state.

Redundancy: Services can be made to tolerate failures by the use of redundant components. Consider the following examples:

There should always be at least two different routes between any two routers in the Internet.

In the Domain Name System, every name table is replicated in at least two different servers.

A database may be replicated in several servers to ensure that the data remains accessible after the failure of any single server; the servers can be designed to

detect faults in their peers; when a fault is detected in one server, clients are redirected to the remaining servers.

Concurrency

Both services and applications provide resources that can be shared by clients in a distributed system. There is therefore a possibility that several clients will attempt to access a shared resource at the same time. For example, a data structure that records bids for an auction may be accessed very frequently when it gets close to the deadline time. The process that manages a shared resource could take one client request at a time. But that approach limits throughput. Therefore services and applications generally allow multiple client requests to be processed concurrently. To make this more concrete, suppose that each resource is encapsulated as an object and that invocations are executed in concurrent threads. In this case it is possible that several threads may be executing concurrently within an object, in which case their operations on the object may conflict with one another and produce inconsistent results.

Transparency

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components. The implications of transparency are a major influence on the design of the system software.

Access transparency enables local and remote resources to be accessed using identical operations.

Location transparency enables resources to be accessed without knowledge of their physical or network location (for example, which building or IP address).

Concurrency transparency enables several processes to operate concurrently using shared resources without interference between them.

Replication transparency enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers. *Failure transparency* enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.

Mobility transparency allows the movement of resources and clients within a system without affecting the operation of users or programs.

Performance transparency allows the system to be reconfigured to improve performance as loads vary.

Scaling transparency allows the system and applications to expand in scale without change to the system structure or the application algorithms.

Quality of service

Once users are provided with the functionality that they require of a service, such as the file service in a distributed system, we can go on to ask about the quality of the service provided. The main nonfunctional properties of systems that affect the quality of the service experienced by clients and users are *reliability*, *security* and *performance*.

Adaptability to meet changing system configurations and resource availability has been recognized as a further important aspect of service quality.

Some applications, including multimedia applications, handle *time-critical data* – streams of data that are required to be processed or transferred from one process to another at a fixed rate. For example, a movie service might consist of a client program that is retrieving a film from a video server and presenting it on the user's screen. For a satisfactory result the successive frames of video need to be displayed to the user within some specified time limits.

In fact, the abbreviation QoS has effectively been commandeered to refer to the ability of systems to meet such deadlines. Its achievement depends upon the

availability of the necessary computing and network resources at the appropriate times. This implies a requirement for the system to provide guaranteed computing and communication resources that are sufficient to enable applications to complete each task on time (for example, the task of displaying a frame of video).

Systems that are intended for use in real-world environments should be designed to function correctly in the widest possible range of circumstances and in the face of many possible difficulties and threats .

Each type of model is intended to provide an abstract, simplified but consistent description of a relevant aspect of distributed system design:

Physical models are the most explicit way in which to describe a system; they capture the hardware composition of a system in terms of the computers (and other devices, such as mobile phones) and their interconnecting networks.

Architectural models describe a system in terms of the computational and communication tasks performed by its computational elements; the computational elements being individual computers or aggregates of them supported by appropriate network interconnections.

Fundamental models take an abstract perspective in order to examine individual aspects of a distributed system. The fundamental models that examine three important aspects of distributed systems: *interaction models*, which consider the structure and sequencing of the communication between the elements of the system; *failure models*, which consider the ways in which a system may fail to operate correctly and; *security models*, which consider how the system is protected against attempts to interfere with its correct operation or to steal its data.

Architectural models

The architecture of a system is its structure in terms of separately specified components and their interrelationships. The overall goal is to ensure that the structure will meet present and likely future demands on it. Major concerns are to make the system reliable, manageable, adaptable and cost-effective. The architectural design of a building has similar aspects – it determines not only its

appearance but also its general structure and architectural style (gothic, neo-classical, modern) and provides a consistent frame of reference for the design.

Software layers

The concept of layering is a familiar one and is closely related to abstraction. In a layered approach, a complex system is partitioned into a number of layers, with a given layer making use of the services offered by the layer below. A given layer therefore offers a software abstraction, with higher layers being unaware of implementation details, or indeed of any other layers beneath them.

In terms of distributed systems, this equates to a vertical organization of services into service layers. A distributed service can be provided by one or more server processes, interacting with each other and with client processes in order to maintain a consistent system-wide view of the service's resources. For example, a network time service is implemented on the Internet based on the Network Time Protocol (NTP) by server processes running on hosts throughout the Internet that supply the current time to any client that requests it and adjust their version of the current time as a result of interactions with each other. Given the complexity of distributed systems, it is often helpful to organize such services into layers. the important terms *platform* and *middleware*, which define as follows:

The important terms *platform* and *middleware*, which is defined as follows:

A platform for distributed systems and applications consists of the lowest-level hardware and software layers. These low-level layers provide services to the layers above them, which are implemented independently in each computer, bringing the system's programming interface up to a level that facilitates communication and coordination between processes. Intel x86/Windows, Intel x86/Solaris, Intel x86/Mac OS X, Intel x86/Linux and ARM/Symbian are major examples.

