Chapter 1

# Introduction

The pace at which computer systems change was, is, and continues to be overwhelming. From 1945, when the modern computer era began, until about 1985, computers were large and expensive. Moreover, for lack of a way to connect them, these computers operated independently from one another.

Starting in the mid-1980s, however, two advances in technology began to change that situation. The first was the development of powerful microprocessors. Initially, these were 8-bit machines, but soon 16-, 32-, and 64-bit CPUs became common. With multicore CPUs, we now are refacing the challenge of adapting and developing programs to exploit parallelism. In any case, the current generation of machines have the computing power of the mainframes deployed 30 or 40 years ago, but for 1/1000th of the price or less.

The second development was the invention of high-speed computer networks. **Local-area networks** or **LANs** allow thousands of machines within a building to be connected in such a way that small amounts of information can be transferred in a few microseconds or so. Larger amounts of data can be moved between machines at rates of billions of *bits per second* (**bps**). **Wide-area network** or **WANs** allow hundreds of millions of machines all over the earth to be connected at speeds varying from tens of thousands to hundreds of millions bps and more.

Parallel to the development of increasingly powerful and networked machines, we have also been able to witness miniaturization of computer systems with perhaps the smartphone as the most impressive outcome. Packed with sensors, lots of memory, and a powerful CPU, these devices are nothing less than full-fledged computers. Of course, they also have networking capabilities. Along the same lines, so-called [plug computer]plug computers are

A version of this chapter has been published as "A Brief Introduction to Distributed Systems," Computing, vol. 98(10):967-1009, 2016.

finding their way to the market. These small computers, often the size of a power adapter, can be plugged directly into an outlet and offer near-desktop performance.

The result of these technologies is that it is now not only feasible, but easy, to put together a computing system composed of a large numbers of networked computers, be they large or small. These computers are generally geographically dispersed, for which reason they are usually said to form a **distributed system**. The size of a distributed system may vary from a handful of devices, to millions of computers. The interconnection network may be wired, wireless, or a combination of both. Moreover, distributed systems are often highly dynamic, in the sense that computers can join and leave, with the topology and performance of the underlying network almost continuously changing.

In this chapter, we provide an initial exploration of distributed systems and their design goals, and follow that up by discussing some well-known types of systems.

## 1.1 What is a distributed system?

Various definitions of distributed systems have been given in the literature, none of them satisfactory, and none of them in agreement with any of the others. For our purposes it is sufficient to give a loose characterization:

> *A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.*

This definition refers to two characteristic features of distributed systems. The first one is that a distributed system is a collection of computing elements each being able to behave independently of each other. A computing element, which we will generally refer to as a **node**, can be either a hardware device or a software process. A second feature is that users (be they people or applications) believe they are dealing with a single system. This means that one way or another the autonomous nodes need to collaborate. How to establish this collaboration lies at the heart of developing distributed systems. Note that we are not making any assumptions concerning the type of nodes. In principle, even within a single system, they could range from high-performance mainframe computers to small devices in sensor networks. Likewise, we make no assumptions concerning the way that nodes are interconnected.

### Characteristic 1: Collection of autonomous computing elements

Modern distributed systems can, and often will, consist of all kinds of nodes, ranging from very big high-performance computers to small plug computers

or even smaller devices. A fundamental principle is that nodes can act independently from each other, although it should be obvious that if they ignore each other, then there is no use in putting them into the same distributed system. In practice, nodes are programmed to achieve common goals, which are realized by exchanging messages with each other. A node reacts to incoming messages, which are then processed and, in turn, leading to further communication through message passing.

An important observation is that, as a consequence of dealing with independent nodes, each one will have its own notion of time. In other words, we cannot always assume that there is something like a **global clock**. This lack of a common reference of time leads to fundamental questions regarding the synchronization and coordination within a distributed system, which we will come to discuss extensively in Chapter 6. The fact that we are dealing with a *collection* of nodes implies that we may also need to manage the membership and organization of that collection. In other words, we may need to register which nodes may or may not belong to the system, and also provide each member with a list of nodes it can directly communicate with.

Managing **group membership** can be exceedingly difficult, if only for reasons of admission control. To explain, we make a distinction between open and closed groups. In an **open group**, any node is allowed to join the distributed system, effectively meaning that it can send messages to any other node in the system. In contrast, with a **closed group**, only the members of that group can communicate with each other and a separate mechanism is needed to let a node join or leave the group.

It is not difficult to see that admission control can be difficult. First, a mechanism is needed to authenticate a node, and as we shall see in Chapter 9, if not properly designed, managing authentication can easily create a scalability bottleneck. Second, each node must, in principle, check if it is indeed communicating with another group member and not, for example, with an intruder aiming to create havoc. Finally, considering that a member can easily communicate with nonmembers, if confidentiality is an issue in the communication within the distributed system, we may be facing trust issues.

overlay network = rede sobreposta

Você pode pensar na rede overlay como uma infraestrutura lógica (ex.: nós organizados em uma árvore) sobre uma infraestrutura física de rede. Isso significa que um enlace na rede overlay (ex.: aresta entre 2 nós da árvore) não precisa corresponder à um enlace físico na rede física

Concerning the organization of the collection, practice shows that a distributed system is often organized as an **overlay network** [Tarkoma, 2010]. In this case, a node is typically a software process equipped with a list of other processes it can directly send messages to. It may also be the case that a neighbor needs to be first looked up. Message passing is then done through TCP/IP or UDP channels, but as we shall see in Chapter 4, higher-level facilities may be available as well. There are roughly two types of overlay networks:

**Structured overlay:** In this case, each node has a well-defined set of neighbors with whom it can communicate. For example, the nodes are organized in a tree or logical ring.

 

**Unstructured overlay:** In these overlays, each node has a number of references to randomly selected other nodes.

In any case, an overlay network should, in principle, always be **connected**, meaning that between any two nodes there is always a communication path allowing those nodes to route messages from one to the other. A well-known class of overlays is formed by **peer-to-peer (P2P) networks**. Examples of overlays will be discussed in detail in Chapter 2 and later chapters. It is important to realize that the organization of nodes requires special effort and that it is sometimes one of the more intricate parts of distributed-systems management.

### Characteristic 2: Single coherent system

As mentioned, a distributed system should appear as a single coherent system. In some cases, researchers have even gone so far as to say that there should be a single-system view, meaning that end users should not even notice that they are dealing with the fact that processes, data, and control are dispersed across a computer network. Achieving a single-system view is often asking too much, for which reason, in our definition of a distributed system, we have opted for something weaker, namely that it *appears* to be coherent. Roughly speaking, a distributed system is coherent if it behaves according to the expectations of its users. More specifically, in a single coherent system the collection of nodes as a whole operates the same, no matter where, when, and how interaction between a user and the system takes place.

Offering a single coherent view is often challenging enough. For example, it requires that an end user would not be able to tell exactly on which computer a process is currently executing, or even perhaps that part of a task has been spawned off to another process executing somewhere else. Likewise, where data is stored should be of no concern, and neither should it matter that the system may be replicating data to enhance performance. This so-called **distribution transparency**, which we will discuss more extensively in Section 1.2, is an important design goal of distributed systems. In a sense, it is akin to the approach taken in many Unix-like operating systems in which resources are accessed through a unifying file-system interface, effectively hiding the differences between files, storage devices, and main memory, but also networks.

However, striving for a single coherent system introduces an important trade-off. As we cannot ignore the fact that a distributed system consists of multiple, networked nodes, it is inevitable that at any time only a part of the system fails. This means that unexpected behavior in which, for example, some applications may continue to execute successfully while others come to a grinding halt, is a reality that needs to be dealt with. Although **partial failures** are inherent to any complex system, in distributed systems they are

 

particularly difficult to hide. It led Turing-Award winner Leslie Lamport, to describe a distributed system as "[...] one in which the failure of a computer you didn't even know existed can render your own computer unusable."

### Middleware and distributed systems

To assist the development of distributed applications, distributed systems are often organized to have a separate layer of software that is logically placed on top of the respective operating systems of the computers that are part of the system. This organization is shown in Figure 1.1, leading to what is known as **middleware** [Bernstein, 1996].

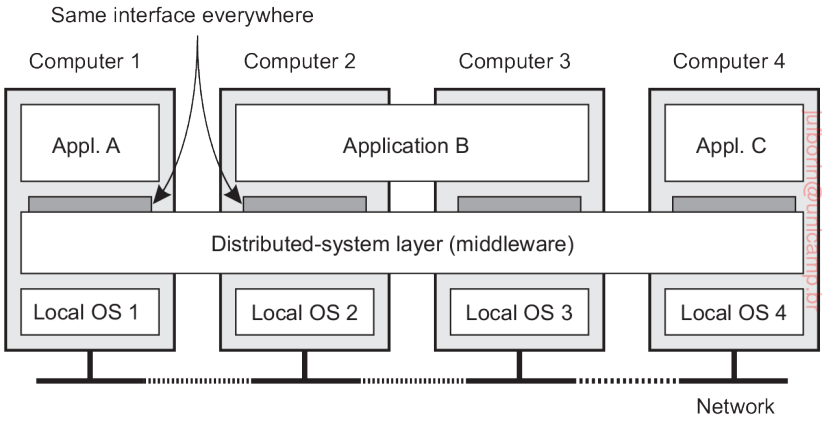


**Figure 1.1:** A distributed system organized in a middleware layer, which extends over multiple machines, offering each application the same interface.

Figure 1.1 shows four networked computers and three applications, of which application B is distributed across computers 2 and 3. Each application is offered the same interface. The distributed system provides the means for components of a single distributed application to communicate with each other, but also to let different applications communicate. At the same time, it hides, as best and reasonably as possible, the differences in hardware and operating systems from each application.

In a sense, middleware is the same to a distributed system as what an operating system is to a computer: a manager of resources offering its applications to efficiently share and deploy those resources across a network. Next to resource management, it offers services that can also be found in most operating systems, including:

- Facilities for interapplication communication.
- Security services.
- Accounting services.
- Masking of and recovery from failures.

 

The main difference with their operating-system equivalents, is that middleware services are offered in a networked environment. Note also that most services are useful to many applications. In this sense, middleware can also be viewed as a container of commonly used components and functions that now no longer have to be implemented by applications separately. To further illustrate these points, let us briefly consider a few examples of typical middleware services.

**Communication:** A common communication service is the so-called **Remote Procedure Call** (**RPC**). An RPC service, to which we return in Chapter 4, allows an application to invoke a function that is implemented and executed on a remote computer as if it was locally available. To this end, a developer need merely specify the function header expressed in a special programming language, from which the RPC subsystem can then generate the necessary code that establishes remote invocations.

**Transactions:** Many applications make use of multiple services that are distributed among several computers. Middleware generally offers special support for executing such services in an all-or-nothing fashion, commonly referred to as an atomic **transaction**. In this case, the application developer need only specify the remote services involved, and by following a standardized protocol, the middleware makes sure that every service is invoked, or none at all.

**Service composition:** It is becoming increasingly common to develop new applications by taking existing programs and gluing them together. This is notably the case for many Web-based applications, in particular those known as **Web services** [Alonso et al., 2004]. Web-based middleware can help by standardizing the way Web services are accessed and providing the means to generate their functions in a specific order. A simple example of how service composition is deployed is formed by **mashups**: Web pages that combine and aggregate data from different sources. Well-known mashups are those based on Google maps in which maps are enhanced with extra information such as trip planners or real-time weather forecasts.

**Reliability:** As a last example, there has been a wealth of research on providing enhanced functions for building reliable distributed applications. The Horus toolkit [van Renesse et al., 1994] allows a developer to build an application as a group of processes such that any message sent by one process is guaranteed to be received by all or no other process. As it turns out, such guarantees can greatly simplify developing distributed applications and are typically implemented as part of the middleware.

**Note 1.1** (Historical note: The term middleware)
Although the term middleware became popular in the mid 1990s, it was most likely mentioned for the first time in a report on a NATO software engineering conference, edited by Peter Naur and Brian Randell in October 1968 [Naur and Randell, 1968]. Indeed, middleware was placed precisely between applications and service routines (the equivalent of operating systems).

## 1.2 Design goals

Just because it is possible to build distributed systems does not necessarily mean that it is a good idea. In this section we discuss four important goals that should be met to make building a distributed system worth the effort. A distributed system should make resources easily accessible; it should hide the fact that resources are distributed across a network; it should be open; and it should be scalable.

### Supporting resource sharing

An important goal of a distributed system is to make it easy for users (and applications) to access and share remote resources. Resources can be virtually anything, but typical examples include peripherals, storage facilities, data, files, services, and networks, to name just a few. There are many reasons for wanting to share resources. One obvious reason is that of economics. For example, it is cheaper to have a single high-end reliable storage facility be shared than having to buy and maintain storage for each user separately.

Connecting users and resources also makes it easier to collaborate and exchange information, as is illustrated by the success of the Internet with its simple protocols for exchanging files, mail, documents, audio, and video. The connectivity of the Internet has allowed geographically widely dispersed groups of people to work together by means of all kinds of **groupware**, that is, software for collaborative editing, teleconferencing, and so on, as is illustrated by multinational software-development companies that have outsourced much of their code production to Asia.

However, resource sharing in distributed systems is perhaps best illustrated by the success of file-sharing peer-to-peer networks like **BitTorrent**. These distributed systems make it extremely simple for users to share files across the Internet. Peer-to-peer networks are often associated with distribution of media files such as audio and video. In other cases, the technology is used for distributing large amounts of data, as in the case of software updates, backup services, and data synchronization across multiple servers.

 

**Note 1.2** (More information: Sharing folders worldwide)
To illustrate where we stand when it comes to seamless integration of resource-sharing facilities in a networked environment, Web-based services are now deployed that allow a group of users to place files into a special shared folder that is maintained by a third party somewhere on the Internet. Using special software, the shared folder is barely distinguishable from other folders on a user's computer. In effect, these services replace the use of a shared directory on a local distributed file system, making data available to users independent of the organization they belong to, and independent of where they are. The service is offered for different operating systems. Where exactly data are stored is completely hidden from the end user.

## Making distribution transparent

An important goal of a distributed system is to hide the fact that its processes and resources are physically distributed across multiple computers possibly separated by large distances. In other words, it tries to make the distribution of processes and resources **transparent**, that is, invisible, to end users and applications.

### Types of distribution transparency

The concept of transparency can be applied to several aspects of a distributed system, of which the most important ones are listed in Figure 1.2. We use the term *object* to mean either a process or a resource.

| Transparency | Description |
|---|---|
| Access | Hide differences in data representation and how an object is accessed |
| Location | Hide where an object is located |
| Relocation | Hide that an object may be moved to another location while in use |
| Migration | Hide that an object may move to another location |
| Replication | Hide that an object is replicated |
| Concurrency | Hide that an object may be shared by several independent users |
| Failure | Hide the failure and recovery of an object |

**Figure 1.2:** Different forms of transparency in a distributed system (see ISO [1995]). An object can be a resource or a process.

**Access transparency** deals with hiding differences in data representation and the way that objects can be accessed. At a basic level, we want to hide

differences in machine architectures, but more important is that we reach agreement on how data is to be represented by different machines and operating systems. For example, a distributed system may have computer systems that run different operating systems, each having their own file-naming conventions. Differences in naming conventions, differences in file operations, or differences in how low-level communication with other processes is to take place, are examples of access issues that should preferably be hidden from users and applications.

An important group of transparency types concerns the location of a process or resource. **Location transparency** refers to the fact that users cannot tell where an object is physically located in the system. Naming plays an important role in achieving location transparency. In particular, location transparency can often be achieved by assigning only logical names to resources, that is, names in which the location of a resource is not secretly encoded. An example of a such a name is the **uniform resource locator** (**URL**) `http://www.prenhall.com/index.html`, which gives no clue about the actual location of Prentice Hall's main Web server. The URL also gives no clue as to whether the file `index.html` has always been at its current location or was recently moved there. For example, the entire site may have been moved from one data center to another, yet users should not notice. The latter is an example of **relocation transparency**, which is becoming increasingly important in the context of cloud computing to which we return later in this chapter.

Where relocation transparency refers to *being* moved by the distributed system, **migration transparency** is offered by a distributed system when it supports the mobility of processes and resources initiated by users, without affecting ongoing communication and operations. A typical example is communication between mobile phones: regardless whether two people are actually moving, mobile phones will allow them to continue their conversation. Other examples that come to mind include online tracking and tracing of goods as they are being transported from one place to another, and teleconferencing (partly) using devices that are equipped with mobile Internet.

As we shall see, replication plays an important role in distributed systems. For example, resources may be replicated to increase availability or to improve performance by placing a copy close to the place where it is accessed. **Replication transparency** deals with hiding the fact that several copies of a resource exist, or that several processes are operating in some form of lockstep mode so that one can take over when another fails. To hide replication from users, it is necessary that all replicas have the same name. Consequently, a system that supports replication transparency should generally support location transparency as well, because it would otherwise be impossible to refer to replicas at different locations.

We already mentioned that an important goal of distributed systems is

 

to allow sharing of resources. In many cases, sharing resources is done in a cooperative way, as in the case of communication channels. However, there are also many examples of competitive sharing of resources. For example, two independent users may each have stored their files on the same file server or may be accessing the same tables in a shared database. In such cases, it is important that each user does not notice that the other is making use of the same resource. This phenomenon is called **concurrency transparency**. An important issue is that concurrent access to a shared resource leaves that resource in a consistent state. Consistency can be achieved through locking mechanisms, by which users are, in turn, given exclusive access to the desired resource. A more refined mechanism is to make use of transactions, but these may be difficult to implement in a distributed system, notably when scalability is an issue.

Last, but certainly not least, it is important that a distributed system provides **failure transparency**. This means that a user or application does not notice that some piece of the system fails to work properly, and that the system subsequently (and automatically) recovers from that failure. Masking failures is one of the hardest issues in distributed systems and is even impossible when certain apparently realistic assumptions are made, as we will discuss in Chapter 8. The main difficulty in masking and transparently recovering from failures lies in the inability to distinguish between a dead process and a painfully slowly responding one. For example, when contacting a busy Web server, a browser will eventually time out and report that the Web page is unavailable. At that point, the user cannot tell whether the server is actually down or that the network is badly congested.

### Degree of distribution transparency

Although distribution transparency is generally considered preferable for any distributed system, there are situations in which attempting to blindly hide all distribution aspects from users is not a good idea. A simple example is requesting your electronic newspaper to appear in your mailbox before 7 AM local time, as usual, while you are currently at the other end of the world living in a different time zone. Your morning paper will not be the morning paper you are used to.

Likewise, a wide-area distributed system that connects a process in San Francisco to a process in Amsterdam cannot be expected to hide the fact that Mother Nature will not allow it to send a message from one process to the other in less than approximately 35 milliseconds. Practice shows that it actually takes several hundred milliseconds using a computer network. Signal transmission is not only limited by the speed of light, but also by limited processing capacities and delays in the intermediate switches.

There is also a trade-off between a high degree of transparency and the performance of a system. For example, many Internet applications repeatedly

Sistemas distribuídos usam as redes de computadores para comunicação entre seus componentes. O tempo de transmissão (latência/delay) de uma mensagem de um ponto até outro tem como limitante inferior a velocidade da luz. Além disso, há outros fatores que podem impactar na latência dessa mensagem e que estão fora do nosso controle. Assim sendo, latência é um obstáculo na provisão de transparência.

try to contact a server before finally giving up. Consequently, attempting to mask a transient server failure before trying another one may slow down the system as a whole. In such a case, it may have been better to give up earlier, or at least let the user cancel the attempts to make contact.

Another example is where we need to guarantee that several replicas, located on different continents, must be consistent all the time. In other words, if one copy is changed, that change should be propagated to all copies before allowing any other operation. It is clear that a single update operation may now even take seconds to complete, something that cannot be hidden from users.

Finally, there are situations in which it is not at all obvious that hiding distribution is a good idea. As distributed systems are expanding to devices that people carry around and where the very notion of location and context awareness is becoming increasingly important, it may be best to actually *expose* distribution rather than trying to hide it. An obvious example is making use of location-based services, which can often be found on mobile phones, such as finding the nearest Chinese take-away or checking whether any of your friends are nearby.

There are also other arguments against distribution transparency. Recognizing that full distribution transparency is simply impossible, we should ask ourselves whether it is even wise to *pretend* that we can achieve it. It may be much better to make distribution explicit so that the user and application developer are never tricked into believing that there is such a thing as transparency. The result will be that users will much better understand the (sometimes unexpected) behavior of a distributed system, and are thus much better prepared to deal with this behavior.

**Note 1.3** (Discussion: Against distribution transparency)
Several researchers have argued that hiding distribution will lead to only further complicating the development of distributed systems, exactly for the reason that full distribution transparency can never be achieved. A popular technique for achieving **a**ccess transparency is to extend procedure calls to remote servers. However, Waldo et al. [1997] already pointed out that attempting to hide distribution by means of such remote procedure calls can lead to poorly understood semantics, for the simple reason that a procedure call *does* change when executed over a faulty communication link.

As an alternative, various researchers and practitioners are now arguing for less transparency, for example, by more explicitly using message-style communication, or more explicitly posting requests to, and getting results from remote machines, as is done in the Web when fetching pages. Such solutions will be discussed in detail in the next chapter.

A somewhat radical standpoint is taken by Wams [2011] by stating that partial failures preclude relying on the successful execution of a remote service. If such reliability cannot be guaranteed, it is then best to always perform only local

 

executions, leading to the **copy-before-use** principle. According to this principle, data can be accessed only after they have been transferred to the machine of the process wanting that data. Moreover, modifying a data item should not be done. Instead, it can only be updated to a new version. It is not difficult to imagine that many other problems will surface. However, Wams shows that many existing applications can be retrofitted to this alternative approach without sacrificing functionality.

The conclusion is that aiming for distribution transparency may be a nice goal when designing and implementing distributed systems, but that it should be considered together with other issues such as performance and comprehensibility. The price for achieving full transparency may be surprisingly high.

## Being open

Another important goal of distributed systems is openness. An **open distributed system** is essentially a system that offers components that can easily be used by, or integrated into other systems. At the same time, an open distributed system itself will often consist of components that originate from elsewhere.

### Interoperability, composability, and extensibility

To be open means that components should adhere to standard rules that describe the syntax and semantics of what those components have to offer (i.e., which service they provide). A general approach is to define services through **interfaces** using an **Interface Definition Language** (**IDL**). Interface definitions written in an IDL nearly always capture only the syntax of services. In other words, they specify precisely the names of the functions that are available together with types of the parameters, return values, possible exceptions that can be raised, and so on. The hard part is specifying precisely what those services do, that is, the semantics of interfaces. In practice, such specifications are given in an informal way by means of natural language.

If properly specified, an interface definition allows an arbitrary process that needs a certain interface, to talk to another process that provides that interface. It also allows two independent parties to build completely different implementations of those interfaces, leading to two separate components that operate in exactly the same way.

Proper specifications are complete and neutral. Complete means that everything that is necessary to make an implementation has indeed been specified. However, many interface definitions are not at all complete, so that it is necessary for a developer to add implementation-specific details.