Tanenbaum, A.S and van Steen, M. "Distributed Systems", 3rd Ed. (v. 3.03), https://www.distributed-systems.net, 2020.

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.

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].

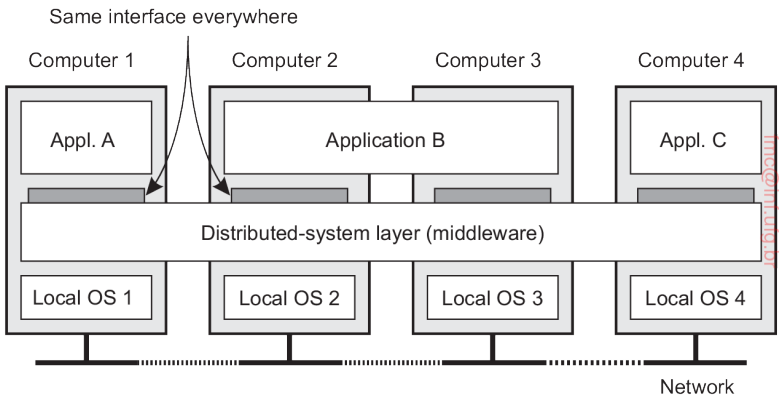

Figure 1.1: A distributed system organized in a middleware layer, which extends over multiple machines, offering each application the same interface.

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

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.

Just as important is the fact that specifications do not prescribe what an implementation should look like; they should be neutral.

As pointed out in Blair and Stefani [1998], completeness and neutrality are important for interoperability and portability. **Interoperability** characterizes the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard. **Portability** characterizes to what extent an application developed for a distributed system A can be executed, without modification, on a different distributed system B that implements the same interfaces as A.

Another important goal for an open distributed system is that it should be easy to configure the system out of different components (possibly from different developers). Also, it should be easy to add new components or replace existing ones without affecting those components that stay in place. In other words, an open distributed system should also be **extensible**. For example, in an extensible system, it should be relatively easy to add parts that run on a different operating system, or even to replace an entire file system.

> **Note 1.4** (Discussion: Open systems in practice)
> Of course, what we have just described is an ideal situation. Practice shows that many distributed systems are not as open as we would like and that still a lot of effort is needed to put various bits and pieces together to make a distributed system. One way out of the lack of openness is to simply reveal all the gory details of a component and to provide developers with the actual source code. This approach is becoming increasingly popular, leading to so-called open source projects where large groups of people contribute to improving and debugging systems. Admittedly, this is as open as a system can get, but whether it is the best way is questionable.

Separating policy from mechanism

To achieve flexibility in open distributed systems, it is crucial that the system be organized as a collection of relatively small and easily replaceable or adaptable components. This implies that we should provide definitions of not only the highest-level interfaces, that is, those seen by users and applications, but also definitions for interfaces to internal parts of the system and describe how those parts interact. This approach is relatively new. Many older and even contemporary systems are constructed using a monolithic approach in which components are only logically separated but implemented as one, huge program. This approach makes it hard to replace or adapt a component without affecting the entire system. Monolithic systems thus tend to be closed instead of open.

The need for changing a distributed system is often caused by a component that does not provide the optimal policy for a specific user or application.

As an example, consider caching in Web browsers. There are many different parameters that need to be considered:

Storage: Where is data to be cached? Typically, there will be an in-memory cache next to storage on disk. In the latter case, the exact position in the local file system needs to be considered.

Exemption: When the cache fills up, which data is to be removed so that newly fetched pages can be stored?

Sharing: Does each browser make use of a private cache, or is a cache to be shared among browsers of different users?

Refreshing: When does a browser check if cached data is still up-to-date? Caches are most effective when a browser can return pages without having to contact the original Web site. However, this bears the risk of returning stale data. Note also that refresh rates are highly dependent on which data is actually cached: whereas timetables for trains hardly change, this is not the case for Web pages showing current highway-traffic conditions, or worse yet, stock prices.

What we need is a separation between policy and mechanism. In the case of Web caching, for example, a browser should ideally provide facilities for only storing documents and at the same time allow users to decide which documents are stored and for how long. In practice, this can be implemented by offering a rich set of parameters that the user can set (dynamically). When taking this a step further, a browser may even offer facilities for plugging in policies that a user has implemented as a separate component.

Note 1.5 (Discussion: Is a strict separation really what we need?)
In theory, strictly separating policies from mechanisms seems to be the way to go. However, there is an important trade-off to consider: the stricter the separation, the more we need to make sure that we offer the appropriate collection of mechanisms. In practice this means that a rich set of features is offered, in turn leading to many configuration parameters. As an example, the popular Firefox browser comes with a few hundred configuration parameters. Just imagine how the configuration space explodes when considering large distributed systems consisting of many components. In other words, strict separation of policies and mechanisms may lead to highly complex configuration problems.

One option to alleviate these problems is to provide reasonable defaults, and this is what often happens in practice. An alternative approach is one in which the system observes its own usage and dynamically changes parameter settings. This leads to what are known as **self-configurable systems**. Nevertheless, the fact alone that many mechanisms need to be offered in order to support a wide range of policies often makes coding distributed systems very complicated. Hard coding policies into a distributed system may reduce complexity considerably, but at the price of less flexibility.

Finding the right balance in separating policies from mechanisms is one of the reasons why designing a distributed system is often more an art than a science.

Being scalable

For many of us, worldwide connectivity through the Internet is as common as being able to send a postcard to anyone anywhere around the world. Moreover, where until recently we were used to having relatively powerful desktop computers for office applications and storage, we are now witnessing that such applications and services are being placed in what has been coined "the cloud," in turn leading to an increase of much smaller networked devices such as tablet computers. With this in mind, scalability has become one of the most important design goals for developers of distributed systems.

Scalability dimensions

Scalability of a system can be measured along at least three different dimensions (see [Neuman, 1994]):

Size scalability: A system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system without any noticeable loss of performance.

Geographical scalability: A geographically scalable system is one in which the users and resources may lie far apart, but the fact that communication delays may be significant is hardly noticed.

Administrative scalability: An administratively scalable system is one that can still be easily managed even if it spans many independent administrative organizations.

Let us take a closer look at each of these three scalability dimensions.

Size scalability. When a system needs to scale, very different types of problems need to be solved. Let us first consider scaling with respect to size. If more users or resources need to be supported, we are often confronted with the limitations of centralized services, although often for very different reasons. For example, many services are centralized in the sense that they are implemented by means of a single **server** running on a specific machine in the distributed system. In a more modern setting, we may have a group of collaborating servers co-located on a cluster of tightly coupled machines physically placed at the same location. The problem with this scheme is obvious: the server, or group of servers, can simply become a bottleneck when it needs to process an increasing number of requests. To illustrate how this

can happen, let us assume that a service is implemented on a single machine. In that case there are essentially three root causes for becoming a bottleneck:

- The computational capacity, limited by the CPUs
- The storage capacity, including the I/O transfer rate
- The network between the user and the centralized service

Let us first consider the computational capacity. Just imagine a service for computing optimal routes taking real-time traffic information into account. It is not difficult to imagine that this may be primarily a compute-bound service requiring several (tens of) seconds to complete a request. If there is only a single machine available, then even a modern high-end system will eventually run into problems if the number of requests increases beyond a certain point.

Likewise, but for different reasons, we will run into problems when having a service that is mainly I/O bound. A typical example is a poorly designed centralized search engine. The problem with content-based search queries is that we essentially need to match a query against an entire data set. Even with advanced indexing techniques, we may still face the problem of having to process a huge amount of data exceeding the main-memory capacity of the machine running the service. As a consequence, much of the processing time will be determined by the relatively slow disk accesses and transfer of data between disk and main memory. Simply adding more or higher-speed disks will prove not to be a sustainable solution as the number of requests continues to increase.

Finally, the network between the user and the service may also be the cause of poor scalability. Just imagine a video-on-demand service that needs to stream high-quality video to multiple users. A video stream can easily require a bandwidth of 8 to 10 Mbps, meaning that if a service sets up point-to-point connections with its customers, it may soon hit the limits of the network capacity of its own outgoing transmission lines.

There are several solutions to attack size scalability which we discuss below after having looked into geographical and administrative scalability.

Note 1.6 (Advanced: Analyzing service capacity)
Size scalability problems for centralized services can be formally analyzed using queuing theory and making a few simplifying assumptions. At a conceptual level, a centralized service can be modeled as the simple queuing system shown in Figure 1.3: requests are submitted to the service where they are queued until further notice. As soon as the process can handle a next request, it fetches it from the queue, does its work, and produces a response. We largely follow Menasce and Almeida [2002] in explaining the performance of a centralized service.

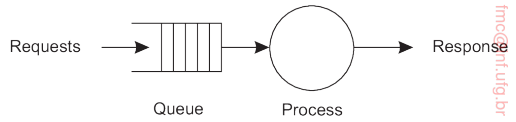

Figure 1.3: A simple model of a service as a queuing system.

In many cases, we may assume that the queue has an infinite capacity, meaning that there is no restriction on the number of requests that can be accepted for further processing. Strictly speaking, this means that the arrival rate of requests is not influenced by what is currently in the queue or being processed. Assuming that the arrival rate of requests is λ requests per second, and that the processing capacity of the service is μ requests per second, one can compute that the fraction of time p_k that there are k requests in the system is equal to:

$$p_k = (1 - \frac{\lambda}{\mu})(\frac{\lambda}{\mu})^k$$

If we define the **utilization** U of a service as the fraction of time that it is busy, then clearly,

$$U = \sum_{k>0} p_k = 1 - p_0 = \frac{\lambda}{\mu} \Rightarrow p_k = (1-U)U^k$$

We can then compute the average number $\overline{N}$ of requests in the system as

$$\overline{N} = \sum_{k\geq 0} k \cdot p_k = \sum_{k\geq 0} k \cdot (1-U)U^k = (1-U)\sum_{k\geq 0} k \cdot U^k = \frac{(1-U)U}{(1-U)^2} = \frac{U}{1-U}.$$

What we are really interested in, is the response time R: how long does it take before the service to process a request, including the time spent in the queue. To that end, we need the average throughput X. Considering that the service is "busy" when at least one request is being processed, and that this then happens with a throughput of μ requests per second, and during a fraction U of the total time, we have:

$$X = \underbrace{U \cdot \mu}_{\text{server at work}} + \underbrace{(1-U) \cdot 0}_{\text{server idle}} = \frac{\lambda}{\mu} \cdot \mu = \lambda$$

Using Little's formula [Trivedi, 2002], we can then derive the response time as

$$R = \frac{\overline{N}}{X} = \frac{S}{1-U} \Rightarrow \frac{R}{S} = \frac{1}{1-U}$$

where $S = \frac{1}{\mu}$, the actual service time. Note that if U is very small, the response-to-service time ratio is close to 1, meaning that a request is virtually instantly processed, and at the maximum speed possible. However, as soon as the utilization comes closer to 1, we see that the response-to-server time ratio quickly increases to very high values, effectively meaning that the system is coming close to a grinding

halt. This is where we see scalability problems emerge. From this simple model, we can see that the only solution is bringing down the service time S. We leave it as an exercise to the reader to explore how S may be decreased.

Geographical scalability. Geographical scalability has its own problems. One of the main reasons why it is still difficult to scale existing distributed systems that were designed for local-area networks is that many of them are based on **synchronous communication**. In this form of communication, a party requesting service, generally referred to as a **client**, blocks until a reply is sent back from the **server** implementing the service. More specifically, we often see a communication pattern consisting of many client-server interactions as may be the case with database transactions. This approach generally works fine in LANs where communication between two machines is often at worst a few hundred microseconds. However, in a wide-area system, we need to take into account that interprocess communication may be hundreds of milliseconds, three orders of magnitude slower. Building applications using synchronous communication in wide-area systems requires a great deal of care (and not just a little patience), notably with a rich interaction pattern between client and server.

Another problem that hinders geographical scalability is that communication in wide-area networks is inherently much less reliable than in local-area networks. In addition, we also need to deal with limited bandwidth. The effect is that solutions developed for local-area networks cannot always be easily ported to a wide-area system. A typical example is streaming video. In a home network, even when having only wireless links, ensuring a stable, fast stream of high-quality video frames from a media server to a display is quite simple. Simply placing that same server far away and using a standard TCP connection to the display will surely fail: bandwidth limitations will instantly surface, but also maintaining the same level of reliability can easily cause headaches.

Yet another issue that pops up when components lie far apart is the fact that wide-area systems generally have only very limited facilities for multipoint communication. In contrast, local-area networks often support efficient broadcasting mechanisms. Such mechanisms have proven to be extremely useful for discovering components and services, which is essential from a management point of view. In wide-area systems, we need to develop separate services, such as naming and directory services to which queries can be sent. These support services, in turn, need to be scalable as well and in many cases no obvious solutions exist as we will encounter in later chapters.

Administrative scalability. Finally, a difficult, and in many cases open, question is how to scale a distributed system across multiple, independent adminis-

trative domains. A major problem that needs to be solved is that of conflicting policies with respect to resource usage (and payment), management, and security.

To illustrate, for many years scientists have been looking for solutions to share their (often expensive) equipment in what is known as a **computational grid**. In these grids, a global distributed system is constructed as a federation of local distributed systems, allowing a program running on a computer at organization A to directly access resources at organization B.

For example, many components of a distributed system that reside within a single domain can often be trusted by users that operate within that same domain. In such cases, system administration may have tested and certified applications, and may have taken special measures to ensure that such components cannot be tampered with. In essence, the users trust their system administrators. However, this trust does not expand naturally across domain boundaries.

Note 1.7 (Example: A modern radio telescope)
As an example, consider developing a modern radio telescope, such as the Pierre Auger Observatory [Abraham et al., 2004]. The final system can be considered as a federated distributed system:

- The radio telescope itself may be a wireless distributed system developed as a grid of a few thousand sensor nodes, each collecting radio signals and collaborating with neighboring nodes to filter out relevant events. The nodes dynamically maintain a sink tree by which selected events are routed to a central point for further analysis.
- The central point needs to be a reasonably powerful system, capable of storing and processing the events sent to it by the sensor nodes. This system is necessarily placed in proximity of the sensor nodes, but is otherwise to be considered to operate independently. Depending on its functionality, it may operate as a small local distributed system. In particular, it stores all recorded events and offers access to remote systems owned by partners in the consortium.
- Most partners have local distributed systems (often in the form of a cluster of computers) that they use to further process the data collected by the telescope. In this case, the local systems directly access the central point at the telescope using a standard communication protocol. Naturally, many results produced within the consortium are made available to each partner.

It is thus seen that the complete system will cross boundaries of several administrative domains, and that special measures are needed to ensure that data that is supposed to be accessible only to (specific) consortium partners cannot be disclosed to unauthorized parties. How to achieve administrative scalability is not obvious.

If a distributed system expands to another domain, two types of security

measures need to be taken. First, the distributed system has to protect itself against malicious attacks from the new domain. For example, users from the new domain may have only read access to the file system in its original domain. Likewise, facilities such as expensive image setters or high-performance computers may not be made available to unauthorized users. Second, the new domain has to protect itself against malicious attacks from the distributed system. A typical example is that of downloading programs such as applets in Web browsers. Basically, the new domain does not know what to expect from such foreign code. The problem, as we shall see in Chapter 9, is how to enforce those limitations.

As a counterexample of distributed systems spanning multiple administrative domains that apparently *do not* suffer from administrative scalability problems, consider modern file-sharing peer-to-peer networks. In these cases, end users simply install a program implementing distributed search and download functions and within minutes can start downloading files. Other examples include peer-to-peer applications for telephony over the Internet such as Skype [Baset and Schulzrinne, 2006], and peer-assisted audio-streaming applications such as Spotify [Kreitz and Niemelä, 2010]. What these distributed systems have in common is that *end users*, and not administrative entities, collaborate to keep the system up and running. At best, underlying administrative organizations such as **Internet Service Providers** (**ISPs**) can police the network traffic that these peer-to-peer systems cause, but so far such efforts have not been very effective.

Scaling techniques

Having discussed some of the scalability problems brings us to the question of how those problems can generally be solved. In most cases, scalability problems in distributed systems appear as performance problems caused by limited capacity of servers and network. Simply improving their capacity (e.g., by increasing memory, upgrading CPUs, or replacing network modules) is often a solution, referred to as **scaling up**. When it comes to **scaling out**, that is, expanding the distributed system by essentially deploying more machines, there are basically only three techniques we can apply: hiding communication latencies, distribution of work, and replication (see also Neuman [1994]).

Hiding communication latencies. **Hiding communication latencies** is applicable in the case of geographical scalability. The basic idea is simple: try to avoid waiting for responses to remote-service requests as much as possible. For example, when a service has been requested at a remote machine, an alternative to waiting for a reply from the server is to do other useful work at the requester's side. Essentially, this means constructing the requesting application in such a way that it uses only **asynchronous communication**. When a reply comes in, the application is interrupted and a special handler is called

to complete the previously issued request. Asynchronous communication can often be used in batch-processing systems and parallel applications in which independent tasks can be scheduled for execution while another task is waiting for communication to complete. Alternatively, a new thread of control can be started to perform the request. Although it blocks waiting for the reply, other threads in the process can continue.

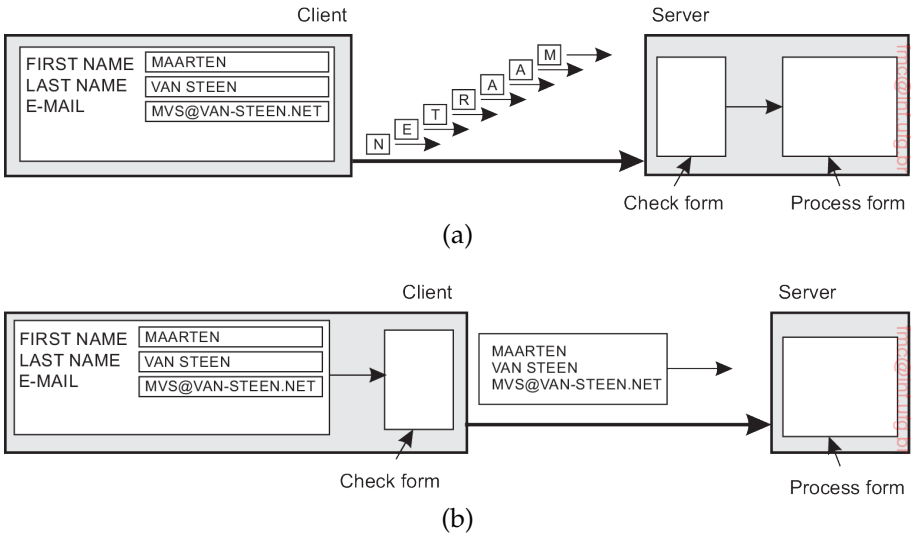

Figure 1.4: The difference between letting (a) a server or (b) a client check forms as they are being filled.

However, there are many applications that cannot make effective use of asynchronous communication. For example, in interactive applications when a user sends a request he will generally have nothing better to do than to wait for the answer. In such cases, a much better solution is to reduce the overall communication, for example, by moving part of the computation that is normally done at the server to the client process requesting the service. A typical case where this approach works is accessing databases using forms. Filling in forms can be done by sending a separate message for each field and waiting for an acknowledgment from the server, as shown in Figure 1.4(a). For example, the server may check for syntactic errors before accepting an entry. A much better solution is to ship the code for filling in the form, and possibly checking the entries, to the client, and have the client return a completed form, as shown in Figure 1.4(b). This approach of shipping code is widely supported by the Web by means of Java applets and Javascript.

Partitioning and distribution. Another important scaling technique is **partitioning and distribution**, which involves taking a component, splitting it

into smaller parts, and subsequently spreading those parts across the system. A good example of partitioning and distribution is the Internet Domain Name System (DNS). The DNS name space is hierarchically organized into a tree of **domains**, which are divided into nonoverlapping **zones**, as shown for the original DNS in Figure 1.5. The names in each zone are handled by a single name server. Without going into too many details now (we return to DNS extensively in Chapter 5), one can think of each path name being the name of a host in the Internet, and is thus associated with a network address of that host. Basically, resolving a name means returning the network address of the associated host. Consider, for example, the name flits.cs.vu.nl. To resolve this name, it is first passed to the server of zone Z1 (see Figure 1.5) which returns the address of the server for zone Z2, to which the rest of name, flits.cs.vu, can be handed. The server for Z2 will return the address of the server for zone Z3, which is capable of handling the last part of the name and will return the address of the associated host.

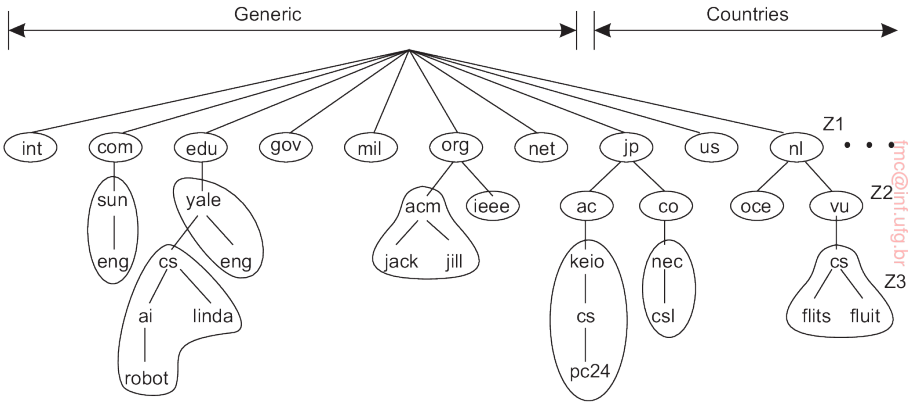

Figure 1.5: An example of dividing the (original) DNS name space into zones.

This examples illustrates how the **naming service** as provided by DNS, is distributed across several machines, thus avoiding that a single server has to deal with all requests for name resolution.

As another example, consider the World Wide Web. To most users, the Web appears to be an enormous document-based information system in which each document has its own unique name in the form of a URL. Conceptually, it may even appear as if there is only a single server. However, the Web is physically partitioned and distributed across *a few hundred million* servers, each handling a number of Web documents. The name of the server handling a document is encoded into that document's URL. It is only because of this distribution of documents that the Web has been capable of scaling to its current size.

Replication. Considering that scalability problems often appear in the form of performance degradation, it is generally a good idea to actually **replicate** components across a distributed system. Replication not only increases availability, but also helps to balance the load between components leading to better performance. Also, in geographically widely dispersed systems, having a copy nearby can hide much of the communication latency problems mentioned before.

Caching is a special form of replication, although the distinction between the two is often hard to make or even artificial. As in the case of replication, caching results in making a copy of a resource, generally in the proximity of the client accessing that resource. However, in contrast to replication, caching is a decision made by the client of a resource and not by the owner of a resource.

There is one serious drawback to caching and replication that may adversely affect scalability. Because we now have multiple copies of a resource, modifying one copy makes that copy different from the others. Consequently, caching and replication leads to **consistency** problems.

To what extent inconsistencies can be tolerated depends highly on the usage of a resource. For example, many Web users find it acceptable that their browser returns a cached document of which the validity has not been checked for the last few minutes. However, there are also many cases in which strong consistency guarantees need to be met, such as in the case of electronic stock exchanges and auctions. The problem with strong consistency is that an update must be immediately propagated to all other copies. Moreover, if two updates happen concurrently, it is often also required that updates are processed in the same order everywhere, introducing an additional global ordering problem. To further aggravate problems, combining consistency with other desirable properties such as availability may simply be impossible, as we discuss in Chapter 8.

Replication therefore often requires some global synchronization mechanism. Unfortunately, such mechanisms are extremely hard or even impossible to implement in a scalable way, if alone because network latencies have a natural lower bound. Consequently, scaling by replication may introduce other, inherently nonscalable solutions. We return to replication and consistency extensively in Chapter 7.

Discussion. When considering these scaling techniques, one could argue that size scalability is the least problematic from a technical point of view. In many cases, increasing the capacity of a machine will save the day, although perhaps there is a high monetary cost to pay. Geographical scalability is a much tougher problem as network latencies are naturally bound from below. As a consequence, we may be forced to copy data to locations close to where clients are, leading to problems of maintaining copies consistent. Practice

shows that combining distribution, replication, and caching techniques with different forms of consistency generally leads to acceptable solutions. Finally, administrative scalability seems to be the most difficult problem to solve, partly because we need to deal with nontechnical issues, such as politics of organizations and human collaboration. The introduction and now widespread use of peer-to-peer technology has successfully demonstrated what can be achieved if end users are put in control [Lua et al., 2005; Oram, 2001]. However, peer-to-peer networks are obviously not the universal solution to all administrative scalability problems.

Pitfalls

It should be clear by now that developing a distributed system is a formidable task. As we will see many times throughout this book, there are so many issues to consider at the same time that it seems that only complexity can be the result. Nevertheless, by following a number of design principles, distributed systems can be developed that strongly adhere to the goals we set out in this chapter.

Distributed systems differ from traditional software because components are dispersed across a network. Not taking this dispersion into account during design time is what makes so many systems needlessly complex and results in flaws that need to be patched later on. Peter Deutsch, at the time working at Sun Microsystems, formulated these flaws as the following false assumptions that many make when developing a distributed application for the first time:

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator

Note how these assumptions relate to properties that are unique to distributed systems: reliability, security, heterogeneity, and topology of the network; latency and bandwidth; transport costs; and finally administrative domains. When developing nondistributed applications, most of these issues will most likely not show up.

Most of the principles we discuss in this book relate immediately to these assumptions. In all cases, we will be discussing solutions to problems that are caused by the fact that one or more assumptions are false. For example, reliable networks simply do not exist and lead to the impossibility of achieving failure transparency. We devote an entire chapter to deal with the fact that

networked communication is inherently insecure. We have already argued that distributed systems need to be open and take heterogeneity into account. Likewise, when discussing replication for solving scalability problems, we are essentially tackling latency and bandwidth problems. We will also touch upon management issues at various points throughout this book.

1.3 Types of distributed systems

Before starting to discuss the principles of distributed systems, let us first take a closer look at the various types of distributed systems. We make a distinction between distributed computing systems, distributed information systems, and pervasive systems (which are naturally distributed).

High performance distributed computing

An important class of distributed systems is the one used for high-performance computing tasks. Roughly speaking, one can make a distinction between two subgroups. In **cluster computing** the underlying hardware consists of a collection of similar workstations or PCs, closely connected by means of a high-speed local-area network. In addition, each node runs the same operating system.

The situation becomes very different in the case of **grid computing**. This subgroup consists of distributed systems that are often constructed as a federation of computer systems, where each system may fall under a different administrative domain, and may be very different when it comes to hardware, software, and deployed network technology.

From the perspective of grid computing, a next logical step is to simply *outsource* the entire infrastructure that is needed for compute-intensive applications. In essence, this is what **cloud computing** is all about: providing the facilities to dynamically construct an infrastructure and compose what is needed from available services. Unlike grid computing, which is strongly associated with high-performance computing, cloud computing is much more than just providing lots of resources. We discuss it briefly here, but will return to various aspects throughout the book.

Note 1.8 (More information: Parallel processing)
High-performance computing more or less started with the introduction of **multiprocessor machines**. In this case, multiple CPUs are organized in such a way that they all have access to the same physical memory, as shown in Figure 1.6(a). In contrast, in a **multicomputer system** several computers are connected through a network and there is no sharing of main memory, as shown in Figure 1.6(b). The shared-memory model proved to be highly convenient for improving the performance of programs and it was relatively easy to program.

Its essence is that multiple threads of control are executing at the same time, while all threads have access to shared data. Access to that data is controlled through well-understood synchronization mechanisms like semaphores (see Ben-Ari [2006] or Herlihy and Shavit [2008] for more information on developing parallel programs). Unfortunately, the model does not easily scale: so far, machines have been developed in which only a few tens (and sometimes hundreds) of CPUs have efficient access to shared memory. To a certain extent, we are seeing the same limitations for multicore processors.

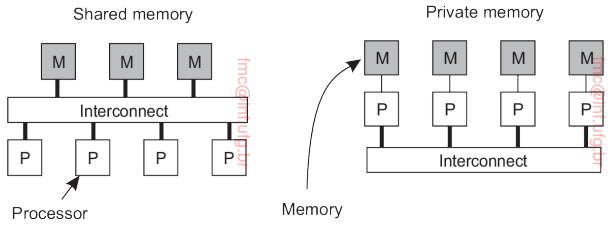

Figure 1.6: A comparison between (a) multiprocessor and (b) multicomputer architectures.

To overcome the limitations of shared-memory systems, high-performance computing moved to distributed-memory systems. This shift also meant that many programs had to make use of message passing instead of modifying shared data as a means of communication and synchronization between threads. Unfortunately, message-passing models have proven to be much more difficult and error-prone compared to the shared-memory programming models. For this reason, there has been significant research in attempting to build so-called **distributed shared-memory multicomputers**, or simply **DSM system** [Amza et al., 1996].

In essence, a DSM system allows a processor to address a memory location at another computer as if it were local memory. This can be achieved using existing techniques available to the operating system, for example, by mapping all main-memory pages of the various processors into a single virtual address space. Whenever a processor A addresses a page located at another processor B, a page fault occurs at A allowing the operating system at A to fetch the content of the referenced page at B in the same way that it would normally fetch it locally from disk. At the same time, processor B would be informed that the page is currently not accessible.

This elegant idea of mimicking shared-memory systems using multicomputers eventually had to be abandoned for the simple reason that performance could never meet the expectations of programmers, who would rather resort to far more intricate, yet better (predictably) performing message-passing programming models.

An important side-effect of exploring the hardware-software boundaries of parallel processing is a thorough understanding of consistency models, to which we return extensively in Chapter 7.

Cluster computing

Cluster computing systems became popular when the price/performance ratio of personal computers and workstations improved. At a certain point, it became financially and technically attractive to build a supercomputer using off-the-shelf technology by simply hooking up a collection of relatively simple computers in a high-speed network. In virtually all cases, cluster computing is used for parallel programming in which a single (compute intensive) program is run in parallel on multiple machines.

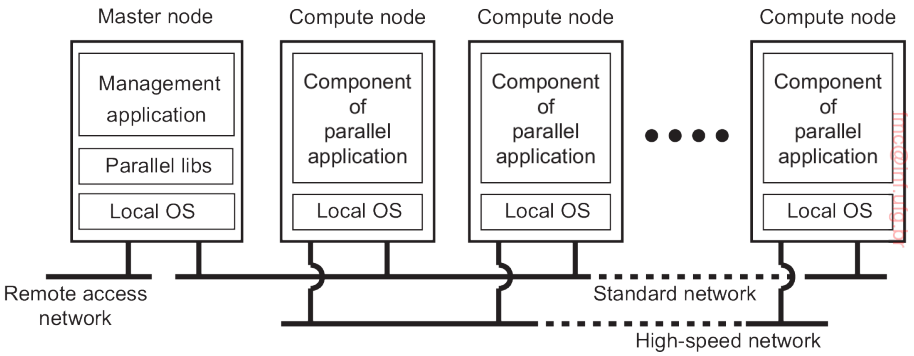

Figure 1.7: An example of a cluster computing system.

One widely applied example of a cluster computer is formed by Linux-based **Beowulf clusters**, of which the general configuration is shown in Figure 1.7. Each cluster consists of a collection of compute nodes that are controlled and accessed by means of a single master node. The master typically handles the allocation of nodes to a particular parallel program, maintains a batch queue of submitted jobs, and provides an interface for the users of the system. As such, the master actually runs the middleware needed for the execution of programs and management of the cluster, while the compute nodes are equipped with a standard operating system extended with typical middleware functions for communication, storage, fault tolerance, and so on. Apart from the master node, the compute nodes are thus seen to be highly identical.

An even more symmetric approach is followed in the **MOSIX system** [Amar et al., 2004]. MOSIX attempts to provide a **single-system image** of a cluster, meaning that to a process a cluster computer offers the ultimate distribution transparency by appearing to be a single computer. As we mentioned, providing such an image under all circumstances is impossible. In the case of MOSIX, the high degree of transparency is provided by allowing processes to dynamically and preemptively migrate between the nodes that make up the cluster. Process migration allows a user to start an application on any node (referred to as the home node), after which it can transparently move to other nodes, for example, to make efficient use of resources. We will return to

process migration in Chapter 3. Similar approaches at attempting to provide a single-system image are compared by [Lottiaux et al., 2005].

However, several modern cluster computers have been moving away from these symmetric architectures to more hybrid solutions in which the middleware is functionally partitioned across different nodes, as explained by Engelmann et al. [2007]. The advantage of such a separation is obvious: having compute nodes with dedicated, lightweight operating systems will most likely provide optimal performance for compute-intensive applications. Likewise, storage functionality can most likely be optimally handled by other specially configured nodes such as file and directory servers. The same holds for other dedicated middleware services, including job management, database services, and perhaps general Internet access to external services.

Grid computing

A characteristic feature of traditional cluster computing is its homogeneity. In most cases, the computers in a cluster are largely the same, have the same operating system, and are all connected through the same network. However, as we just discussed, there has been a trend towards more hybrid architectures in which nodes are specifically configured for certain tasks. This diversity is even more prevalent in **grid-computing systems**: no assumptions are made concerning similarity of hardware, operating systems, networks, administrative domains, security policies, etc.

A key issue in a grid-computing system is that resources from different organizations are brought together to allow the collaboration of a group of people from different institutions, indeed forming a federation of systems. Such a collaboration is realized in the form of a **virtual organization**. The processes belonging to the same virtual organization have access rights to the resources that are provided to that organization. Typically, resources consist of compute servers (including supercomputers, possibly implemented as cluster computers), storage facilities, and databases. In addition, special networked devices such as telescopes, sensors, etc., can be provided as well.

Given its nature, much of the software for realizing grid computing evolves around providing access to resources from different administrative domains, and to only those users and applications that belong to a specific virtual organization. For this reason, focus is often on architectural issues. An architecture initially proposed by Foster et al. [2001] is shown in Figure 1.8, which still forms the basis for many grid computing systems.

The architecture consists of four layers. The lowest *fabric layer* provides interfaces to local resources at a specific site. Note that these interfaces are tailored to allow sharing of resources within a virtual organization. Typically, they will provide functions for querying the state and capabilities of a resource, along with functions for actual resource management (e.g., locking resources).

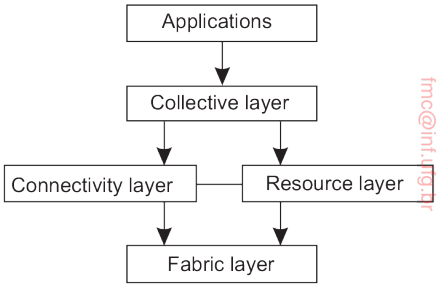

Figure 1.8: A layered architecture for grid computing systems.

The *connectivity layer* consists of communication protocols for supporting grid transactions that span the usage of multiple resources. For example, protocols are needed to transfer data between resources, or to simply access a resource from a remote location. In addition, the connectivity layer will contain security protocols to authenticate users and resources. Note that in many cases human users are not authenticated; instead, programs acting on behalf of the users are authenticated. In this sense, delegating rights from a user to programs is an important function that needs to be supported in the connectivity layer. We return to delegation when discussing security in distributed systems in Chapter 9.

The *resource layer* is responsible for managing a single resource. It uses the functions provided by the connectivity layer and calls directly the interfaces made available by the fabric layer. For example, this layer will offer functions for obtaining configuration information on a specific resource, or, in general, to perform specific operations such as creating a process or reading data. The resource layer is thus seen to be responsible for access control, and hence will rely on the authentication performed as part of the connectivity layer.

The next layer in the hierarchy is the *collective layer*. It deals with handling access to multiple resources and typically consists of services for resource discovery, allocation and scheduling of tasks onto multiple resources, data replication, and so on. Unlike the connectivity and resource layer, each consisting of a relatively small, standard collection of protocols, the collective layer may consist of many different protocols reflecting the broad spectrum of services it may offer to a virtual organization.

Finally, the *application layer* consists of the applications that operate within a virtual organization and which make use of the grid computing environment.

Typically the collective, connectivity, and resource layer form the heart of what could be called a grid middleware layer. These layers jointly provide access to and management of resources that are potentially dispersed across multiple sites.

An important observation from a middleware perspective is that in grid

computing the notion of a site (or administrative unit) is common. This prevalence is emphasized by the gradual shift toward a **service-oriented architecture** in which sites offer access to the various layers through a collection of Web services [Joseph et al., 2004]. This, by now, has led to the definition of an alternative architecture known as the **Open Grid Services Architecture** (**OGSA**) [Foster et al., 2006]. OGSA is based upon the original ideas as formulated by Foster et al. [2001], yet having gone through a standardization process makes it complex, to say the least. OGSA implementations generally follow Web service standards.

Cloud computing

While researchers were pondering on how to organize computational grids that were easily accessible, organizations in charge of running data centers were facing the problem of opening up their resources to customers. Eventually, this led to the concept of **utility computing** by which a customer could upload tasks to a data center and be charged on a per-resource basis. Utility computing formed the basis for what is now called **cloud computing**.

Following Vaquero et al. [2008], cloud computing is characterized by an easily usable and accessible pool of *virtualized* resources. Which and how resources are used can be configured dynamically, providing the basis for scalability: if more work needs to be done, a customer can simply acquire more resources. The link to utility computing is formed by the fact that cloud computing is generally based on a pay-per-use model in which guarantees are offered by means of customized **service-level agreements** (**SLAs**).

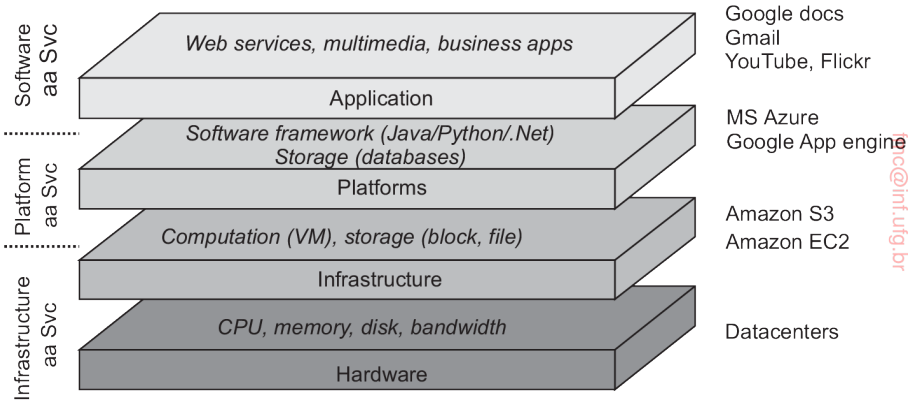

Figure 1.9: The organization of clouds (adapted from Zhang et al. [2010]).

In practice, clouds are organized into four layers, as shown in Figure 1.9 (see also Zhang et al. [2010]):

Hardware: The lowest layer is formed by the means to manage the necessary hardware: processors, routers, but also power and cooling systems. It is generally implemented at data centers and contains the resources that customers normally never get to see directly.

Infrastructure: This is an important layer forming the backbone for most cloud computing platforms. It deploys virtualization techniques (discussed in Section 3.2) to provide customers an infrastructure consisting of virtual storage and computing resources. Indeed, nothing is what it seems: cloud computing evolves around allocating and managing virtual storage devices and virtual servers.

Platform: One could argue that the platform layer provides to a cloud-computing customer what an operating system provides to application developers, namely the means to easily develop and deploy applications that need to run in a cloud. In practice, an application developer is offered a vendor-specific API, which includes calls to uploading and executing a program in that vendor's cloud. In a sense, this is comparable the Unix exec family of system calls, which take an executable file as parameter and pass it to the operating system to be executed.

Also like operating systems, the platform layer provides higher-level abstractions for storage and such. For example, as we discuss in more detail later, the **Amazon S3 storage system** [Murty, 2008] is offered to the application developer in the form of an API allowing (locally created) files to be organized and stored in **buckets**. A bucket is somewhat comparable to a directory. By storing a file in a bucket, that file is automatically uploaded to the Amazon cloud.

Application: Actual applications run in this layer and are offered to users for further customization. Well-known examples include those found in office suites (text processors, spreadsheet applications, presentation applications, and so on). It is important to realize that these applications are again executed in the vendor's cloud. As before, they can be compared to the traditional suite of applications that are shipped when installing an operating system.

Cloud-computing providers offer these layers to their customers through various interfaces (including command-line tools, programming interfaces, and Web interfaces), leading to three different types of services:

- **Infrastructure-as-a-Service** (**IaaS**) covering the hardware and infrastructure layer.
- **Platform-as-a-Service** (**PaaS**) covering the platform layer.
- **Software-as-a-Service** (**SaaS**) in which their applications are covered.

As of now, making use of clouds is relatively easy, and we discuss in later chapters more concrete examples of interfaces to cloud providers. As a consequence, cloud computing as a means for outsourcing local computing infrastructures has become a serious option for many enterprises. However, there are still a number of serious obstacles including provider lock-in, security and privacy issues, and dependency on the availability of services, to mention a few (see also Armbrust et al. [2010]). Also, because the details on how specific cloud computations are actually carried out are generally hidden, and even perhaps unknown or unpredictable, meeting performance demands may be impossible to arrange in advance. On top of this, Li et al. [2010] have shown that different providers may easily show very different performance profiles. Cloud computing is no longer a hype, and certainly a serious alternative to maintaining huge local infrastructures, yet there is still a lot of room for improvement.

Note 1.9 (Advanced: Is cloud computing cheaper?)
One of the important reasons to migrate to a cloud environment is that it may be much cheaper compared to maintaining a local computing infrastructure. There are many ways to compute the savings, but as it turns out, only for simple and obvious cases will straightforward computations give a realistic perspective. Hajjat et al. [2010] propose a more thorough approach, taking into account that part of an application suite is migrated to a cloud, and the other part continues to be operated on a local infrastructure. The crux of their method is providing the right model of a suite of enterprise applications.

The core of their approach is formed by a potentially large set of *software components*. Each enterprise application is assumed to consist of components. Furthermore, each component $\mathsf{C_i}$ is considered to be run on N_i servers. A simple example is a database component to be executed by a single server. A more elaborate example is a Web application for computing bicycle routes, consisting of a Web server front end for rendering HTML pages and accepting user input, a component for computing shortest paths (perhaps under different constraints), and a database component containing various maps.

Each application is modeled as a directed graph, in which a vertex represents a component and an arc $\langle \overrightarrow{i,j} \rangle$ the fact that data flows from component $\mathsf{C_i}$ to component $\mathsf{C_j}$. Each arc has two associated weights: $T_{i,j}$ represents the number of transactions per time unit leading to data flowing from $\mathsf{C_i}$ to $\mathsf{C_j}$, and $S_{i,j}$ the average size of those transactions (i.e., the average amount of data per transaction). They assume that $T_{i,j}$ and $S_{i,j}$ are known, typically obtained through straightforward measurements.

Migrating a suite of applications from a local infrastructure to the cloud then boils down to finding an optimal migration plan M: figuring out for each component $\mathsf{C_i}$, how many n_i of its N_i servers should be moved to the cloud, such that the monetary benefits resulting from M, reduced by the additional costs for communicating over the Internet, are maximal. A plan M should also meet the following constraints:

1. Policy constraints are met. For example, there may be data that is legally required to be located at an organization's local infrastructure.
2. Because communication is now partly across long-haul Internet links, it may be that certain transactions between components become much slower. A plan M is acceptable only if any additional latencies do not violate specific delay constraints.
3. Flow balance equations should be respected: transactions continue to operate correctly, and requests or data are not lost during a transaction.

Let us now look into the benefits and Internet costs of a migration plan.

Benefits For each migration plan M, one can expect to have monetary savings expressed as $\mathit{Benefits}(\mathsf{M})$, because fewer machines or network connections need to be maintained. In many organizations, such costs are known so that it may be relatively simple to compute the savings. On the other hand, there are costs to be made for using the cloud. Hajjat et al. [2010] make a simplifying distinction between the benefit B_c of migrating a compute-intensive component, and the benefit B_s of migrating a storage-intensive component. If there are M_c compute-intensive and M_s storage-intensive components, we have $\mathit{Benefits}(\mathsf{M}) = B_c \cdot M_c + B_s \cdot M_s$. Obviously, much more sophisticated models can be deployed as well.

Internet costs To compute the increased communication costs because components are spread across the cloud as well as the local infrastructure, we need to take user-initiated requests into account. To simplify matters, we make no distinction between internal users (i.e., members of the enterprise), and external users (as one would see in the case of Web applications). Traffic from users before migration can be expressed as:

$$Tr_{local,inet} = \sum_{C_i} (T_{user,i} S_{user,i} + T_{i,user} S_{i,user})$$

where $T_{user,i}$ denotes the number of transactions per time unit leading to data flowing from users to $\mathsf{C_i}$. We have analogous interpretations for $T_{i,user}$, S_{user_i}, and $S_{i,user}$.

For each component $\mathsf{C_i}$, let $C_{i,local}$ denote the servers that continue to operate on the local infrastructure, and $C_{i,cloud}$ its servers that are placed in the cloud. Note that $|C_{i,cloud}| = n_i$. For simplicity, assume that a server from $C_{i,local}$ distributes traffic in the same proportions as a server from $C_{i,cloud}$. We are interested in the rate of transactions between local servers, cloud servers, and between local and cloud servers, after migration. Let $\mathsf{s_k}$ be the server for component $\mathsf{C_k}$ and denote by f_k the fraction n_k / N_k. We then have for the rate of transactions $T^*_{i,j}$ after migration:

$$T^*_{i,j} = \begin{cases} (1-f_i) \cdot (1-f_j) \cdot T_{i,j} & \text{when } \mathsf{s_i} \in C_{i,local} \text{ and } \mathsf{s_j} \in C_{j,local} \\ (1-f_i) \cdot f_j \cdot T_{i,j} & \text{when } \mathsf{s_i} \in C_{i,local} \text{ and } \mathsf{s_j} \in C_{j,cloud} \\ f_i \cdot (1-f_j) \cdot T_{i,j} & \text{when } \mathsf{s_i} \in C_{i,cloud} \text{ and } \mathsf{s_j} \in C_{j,local} \\ f_i \cdot f_j \cdot T_{i,j} & \text{when } \mathsf{s_i} \in C_{i,cloud} \text{ and } \mathsf{s_j} \in C_{j,cloud} \end{cases}$$

$S^*_{i,j}$ is the amount of data associated with $T^*_{i,j}$. Note that f_k denotes the fraction of servers of component C_k that are moved to the cloud. In other words, $(1 - f_k)$ is the fraction that stays in the local infrastructure. We leave it to the reader to give an expression for $T^*_{i,user}$.

Finally, let $cost_{local,inet}$ and $cost_{cloud,inet}$ denote the per-unit Internet costs for traffic to and from the local infrastructure and cloud, respectively. Ignoring a few subtleties explained in [Hajjat et al., 2010], we can then compute the local Internet traffic after migration as:

$$Tr^*_{local,inet} = \sum_{C_{i,local},C_{j,local}} (T^*_{i,j}S^*_{i,j} + T^*_{j,i}S^*_{j,i}) + \sum_{C_{j,local}} (T^*_{user,j}S^*_{user,j} + T^*_{j,user}S^*_{j,user})$$

and, likewise, for the cloud Internet traffic after migration:

$$Tr^*_{cloud,inet} = \sum_{C_{i,cloud},C_{j,cloud}} (T^*_{i,j}S^*_{i,j} + T^*_{j,i}S^*_{j,i}) + \sum_{C_{j,cloud}} (T^*_{user,j}S^*_{user,j} + T^*_{j,user}S^*_{j,user})$$

Together, this leads to a model for the increase in Internet communication costs:

$$cost_{local,inet}(Tr^*_{local,inet} - Tr_{local,inet}) + cost_{cloud,inet}Tr^*_{cloud,inet}$$

Clearly, answering the question whether moving to the cloud is cheaper requires a lot of detailed information and careful planning of exactly what to migrate. Hajjat et al. [2010] provide a first step toward making an informed decision. Their model is more detailed than we are willing to explain here. An important aspect that we have not touched upon is that migrating components also means that special attention will have to be paid to migrating security components. The interested reader is referred to their paper.

Distributed information systems

Another important class of distributed systems is found in organizations that were confronted with a wealth of networked applications, but for which interoperability turned out to be a painful experience. Many of the existing middleware solutions are the result of working with an infrastructure in which it was easier to integrate applications into an **enterprise-wide information system** [Alonso et al., 2004; Bernstein, 1996; Hohpe and Woolf, 2004].

We can distinguish several levels at which integration can take place. In many cases, a networked application simply consists of a server running that application (often including a database) and making it available to remote programs, called **clients**. Such clients send a request to the server for executing a specific operation, after which a response is sent back. Integration at the lowest level allows clients to wrap a number of requests, possibly for different servers, into a single larger request and have it executed as a **distributed transaction**. The key idea is that all, or none of the requests are executed.

As applications became more sophisticated and were gradually separated

into independent components (notably distinguishing database components from processing components), it became clear that integration should also take place by letting applications communicate directly with each other. This has now led to a huge industry that concentrates on **enterprise application integration** (**EAI**).

Distributed transaction processing

To clarify our discussion, we concentrate on database applications. In practice, operations on a database are carried out in the form of **transactions**. Programming using transactions requires special primitives that must either be supplied by the underlying distributed system or by the language runtime system. Typical examples of transaction primitives are shown in Figure 1.10. The exact list of primitives depends on what kinds of objects are being used in the transaction [Gray and Reuter, 1993; Bernstein and Newcomer, 2009]. In a mail system, there might be primitives to send, receive, and forward mail. In an accounting system, they might be quite different. READ and WRITE are typical examples, however. Ordinary statements, procedure calls, and so on, are also allowed inside a transaction. In particular, **remote procedure calls** (**RPC**), that is, procedure calls to remote servers, are often also encapsulated in a transaction, leading to what is known as a **transactional RPC**. We discuss RPCs extensively in Section 4.2.

Primitive	Description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

Figure 1.10: Example primitives for transactions.

BEGIN_TRANSACTION and END_TRANSACTION are used to delimit the scope of a transaction. The operations between them form the body of the transaction. The characteristic feature of a transaction is either all of these operations are executed or none are executed. These may be system calls, library procedures, or bracketing statements in a language, depending on the implementation.

This all-or-nothing property of transactions is one of the four characteristic properties that transactions have. More specifically, transactions adhere to the so-called **ACID** properties:

- **Atomic**: To the outside world, the transaction happens indivisibly
- **Consistent**: The transaction does not violate system invariants

- **Isolated**: Concurrent transactions do not interfere with each other
- **Durable**: Once a transaction commits, the changes are permanent

In distributed systems, transactions are often constructed as a number of subtransactions, jointly forming a **nested transaction** as shown in Figure 1.11. The top-level transaction may fork off children that run in parallel with one another, on different machines, to gain performance or simplify programming. Each of these children may also execute one or more subtransactions, or fork off its own children.

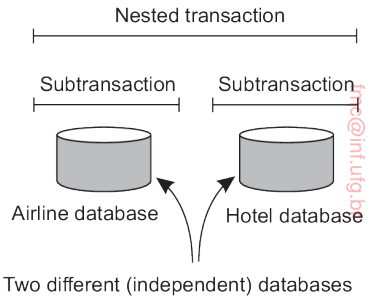

Figure 1.11: A nested transaction.

Subtransactions give rise to a subtle, but important, problem. Imagine that a transaction starts several subtransactions in parallel, and one of these commits, making its results visible to the parent transaction. After further computation, the parent aborts, restoring the entire system to the state it had before the top-level transaction started. Consequently, the results of the subtransaction that committed must nevertheless be undone. Thus the permanence referred to above applies only to top-level transactions.

Since transactions can be nested arbitrarily deep, considerable administration is needed to get everything right. The semantics are clear, however. When any transaction or subtransaction starts, it is conceptually given a private copy of all data in the entire system for it to manipulate as it wishes. If it aborts, its private universe just vanishes, as if it had never existed. If it commits, its private universe replaces the parent's universe. Thus if a subtransaction commits and then later a new subtransaction is started, the second one sees the results produced by the first one. Likewise, if an enclosing (higher level) transaction aborts, all its underlying subtransactions have to be aborted as well. And if several transactions are started concurrently, the result is as if they ran sequentially in some unspecified order.

Nested transactions are important in distributed systems, for they provide a natural way of distributing a transaction across multiple machines. They follow a *logical* division of the work of the original transaction. For example, a transaction for planning a trip by which three different flights need to be

reserved can be logically split up into three subtransactions. Each of these subtransactions can be managed separately and independently of the other two.

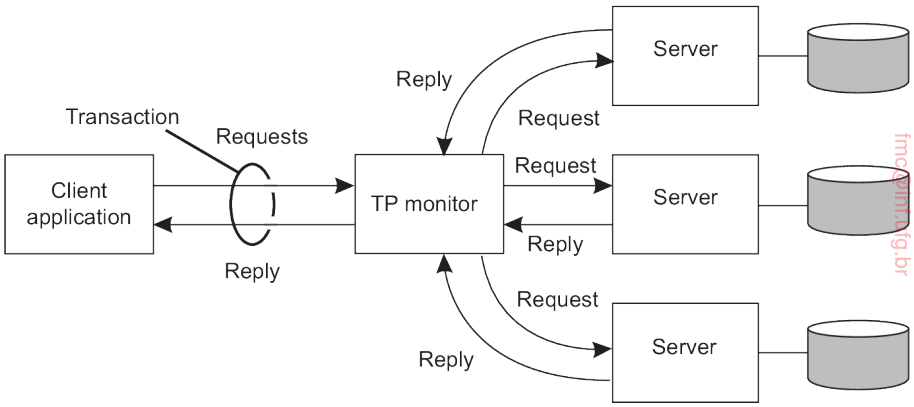

Figure 1.12: The role of a TP monitor in distributed systems.

In the early days of enterprise middleware systems, the component that handled distributed (or nested) transactions formed the core for integrating applications at the server or database level. This component was called a **transaction-processing monitor** or **TP monitor** for short. Its main task was to allow an application to access multiple server/databases by offering it a transactional programming model, as shown in Figure 1.12. Essentially, the TP monitor coordinated the commitment of subtransactions following a standard protocol known as **distributed commit**, which we discuss in Section 8.5.

An important observation is that applications wanting to coordinate several subtransactions into a single transaction did not have to implement this coordination themselves. By simply making use of a TP monitor, this coordination was done for them. This is exactly where middleware comes into play: it implements services that are useful for many applications avoiding that such services have to be reimplemented over and over again by application developers.

Enterprise application integration

As mentioned, the more applications became decoupled from the databases they were built upon, the more evident it became that facilities were needed to integrate applications independently from their databases. In particular, application components should be able to communicate directly with each other and not merely by means of the request/reply behavior that was supported by transaction processing systems.

This need for interapplication communication led to many different communication models, The main idea was that existing applications could directly exchange information, as shown in Figure 1.13.

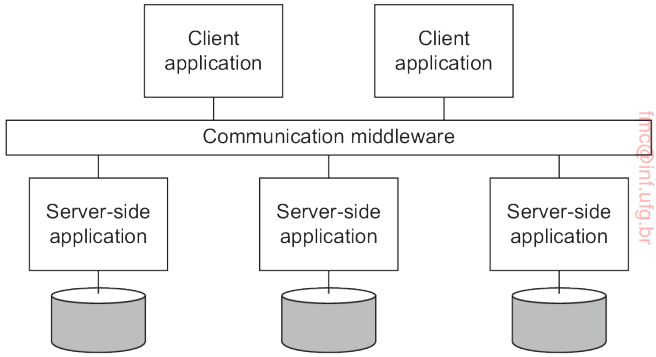

Figure 1.13: Middleware as a communication facilitator in enterprise application integration.

Several types of communication middleware exist. With **remote procedure calls** (**RPC**), an application component can effectively send a request to another application component by doing a local procedure call, which results in the request being packaged as a message and sent to the callee. Likewise, the result will be sent back and returned to the application as the result of the procedure call.

As the popularity of object technology increased, techniques were developed to allow calls to remote objects, leading to what is known as **remote method invocations** (**RMI**). An RMI is essentially the same as an RPC, except that it operates on objects instead of functions.

RPC and RMI have the disadvantage that the caller and callee both need to be up and running at the time of communication. In addition, they need to know exactly how to refer to each other. This tight coupling is often experienced as a serious drawback, and has lead to what is known as **message-oriented middleware**, or simply **MOM**. In this case, applications send messages to logical contact points, often described by means of a subject. Likewise, applications can indicate their interest for a specific type of message, after which the communication middleware will take care that those messages are delivered to those applications. These so-called **publish-subscribe** systems form an important and expanding class of distributed systems.

Note 1.10 (More information: On integrating applications)
Supporting enterprise application integration is an important goal for many mid-

dleware products. In general, there are four ways to integrate applications [Hohpe and Woolf, 2004]:

File transfer: The essence of integration through file transfer, is that an application produces a file containing shared data that is subsequently read by other applications. The approach is technically very simple, making it appealing. The drawback, however, is that there are a lot of things that need to be agreed upon:

- File format and layout: text, binary, its structure, and so on. Nowadays, XML has become popular as its files are, in principle, self-describing.
- File management: where are they stored, how are they named, who is responsible for deleting files?
- Update propagation: When an application produces a file, there may be several applications that need to read that file in order to provide the view of a single coherent system, as we argued in Section 1.1. As a consequence, sometimes separate programs need to be implemented that notify applications of file updates.

Shared database: Many of the problems associated with integration through files are alleviated when using a shared database. All applications will have access to the same data, and often through a high-level language such as SQL. Also, it is easy to notify applications when changes occur, as triggers are often part of modern databases. There are, however, two major drawbacks. First, there is still a need to design a common data schema, which may be far from trivial if the set of applications that need to be integrated is not completely known in advance. Second, when there are many reads and updates, a shared database can easily become a performance bottleneck.

Remote procedure call: Integration through files or a database implicitly assumes that changes by one application can easily trigger other applications to take action. However, practice shows that sometimes small changes should actually trigger many applications to take actions. In such cases, it is not really the change of data that is important, but the execution of a series of actions.

Series of actions are best captured through the execution of a procedure (which may, in turn, lead to all kinds of changes in shared data). To prevent that every application needs to know all the internals of those actions (as implemented by another application), standard encapsulation techniques should be used, as deployed with traditional procedure calls or object invocations. For such situations, an application can best offer a procedure to other applications in the form of a remote procedure call, or RPC. In essence, an RPC allows an application A to make use of the information available only to application B, without giving A direct access to that information. There are many advantages and disadvantages to remote procedure calls, which are discussed in depth in Chapter 4.

Messaging: A main drawback of RPCs is that caller and callee need to be up and running at the same time in order for the call to succeed. However, in

many scenarios this simultaneous activity is often difficult or impossible to guarantee. In such cases, offering a messaging system carrying requests from application A to perform an action at application B, is what is needed. The messaging system ensures that eventually the request is delivered, and if needed, that a response is eventually returned as well. Obviously, messaging is not the panacea for application integration: it also introduces problems concerning data formatting and layout, it requires an application to know where to send a message to, there need to be scenarios for dealing with lost messages, and so on. Like RPCs, we will be discussing these issues extensively in Chapter 4.

What these four approaches tell us, is that application integration will generally not be simple. Middleware (in the form of a distributed system), however, can significantly help in integration by providing the right facilities such as support for RPCs or messaging. As said, enterprise application integration is an important target field for many middleware products.

Pervasive systems

The distributed systems discussed so far are largely characterized by their stability: nodes are fixed and have a more or less permanent and high-quality connection to a network. To a certain extent, this stability is realized through the various techniques for achieving distribution transparency. For example, there are many ways how we can create the illusion that only occasionally components may fail. Likewise, there are all kinds of means to hide the actual network location of a node, effectively allowing users and applications to believe that nodes stay put.

However, matters have changed since the introduction of mobile and embedded computing devices, leading to what are generally referred to as **pervasive systems**. As its name suggests, pervasive systems are intended to naturally blend into our environment. They are naturally also distributed systems, and certainly meet the characterization we gave in Section 1.1.

What makes them unique in comparison to the computing and information systems described so far, is that the separation between users and system components is much more blurred. There is often no single dedicated interface, such as a screen/keyboard combination. Instead, a pervasive system is often equipped with many **sensors** that pick up various aspects of a user's behavior. Likewise, it may have a myriad of **actuators** to provide information and feedback, often even purposefully aiming to *steer* behavior.

Many devices in pervasive systems are characterized by being small, battery-powered, mobile, and having only a wireless connection, although not all these characteristics apply to all devices. These are not necessarily restrictive characteristics, as is illustrated by smartphones [Roussos et al., 2005] and their role in what is now coined as the **Internet of Things** [Mattern and

Floerkemeier, 2010; Stankovic, 2014]. Nevertheless, notably the fact that we often need to deal with the intricacies of wireless and mobile communication, will require special solutions to make a pervasive system as transparent or unobtrusive as possible.

In the following, we make a distinction between three different types of pervasive systems, although there is considerable overlap between the three types: ubiquitous computing systems, mobile systems, and sensor networks. This distinction allows us to focus on different aspects of pervasive systems.

Ubiquitous computing systems

So far, we have been talking about pervasive systems to emphasize that its elements have spread through in many parts of our environment. In a ubiquitous computing system we go one step further: the system is pervasive and continuously present. The latter means that a user will be continuously interacting with the system, often not even being aware that interaction is taking place. Poslad [2009] describes the core requirements for a **ubiquitous computing system** roughly as follows:

1. (**Distribution**) Devices are networked, distributed, and accessible in a transparent manner
2. (**Interaction**) Interaction between users and devices is highly unobtrusive
3. (**Context awareness**) The system is aware of a user's context in order to optimize interaction
4. (**Autonomy**) Devices operate autonomously without human intervention, and are thus highly self-managed
5. (**Intelligence**) The system as a whole can handle a wide range of dynamic actions and interactions

Let us briefly consider these requirements from a distributed-systems perspective.

Ad. 1: Distribution. As mentioned, a ubiquitous computing system is an example of a distributed system: the devices and other computers forming the nodes of a system are simply networked and work together to form the illusion of a single coherent system. Distribution also comes naturally: there will be devices close to users (such as sensors and actuators), connected to computers hidden from view and perhaps even operating remotely in a cloud. Most, if not all, of the requirements regarding distribution transparency mentioned in Section 1.2, should therefore hold.

Ad. 2: Interaction. When it comes to interaction with users, ubiquitous computing systems differ a lot in comparison to the systems we have been

discussing so far. End users play a prominent role in the design of ubiquitous systems, meaning that special attention needs to be paid to how the interaction between users and core system takes place. For ubiquitous computing systems, much of the interaction by humans will be implicit, with an **implicit action** being defined as one "that is not primarily aimed to interact with a computerized system but which such a system understands as input" [Schmidt, 2000]. In other words, a user could be mostly unaware of the fact that input is being provided to a computer system. From a certain perspective, ubiquitous computing can be said to seemingly *hide* interfaces.

A simple example is where the settings of a car's driver's seat, steering wheel, and mirrors is fully personalized. If Bob takes a seat, the system will recognize that it is dealing with Bob and subsequently makes the appropriate adjustments. The same happens when Alice uses the car, while an unknown user will be steered toward making his or her own adjustments (to be remembered for later). This example already illustrates an important role of sensors in ubiquitous computing, namely as input devices that are used to identify a situation (a specific person apparently wanting to drive), whose input analysis leads to actions (making adjustments). In turn, the actions may lead to natural reactions, for example that Bob slightly changes the seat settings. The system will have to take all (implicit and explicit) actions by the user into account and react accordingly.

Ad. 3: Context awareness. Reacting to the sensory input, but also the explicit input from users is more easily said than done. What a ubiquitous computing system needs to do, is to take the *context* in which interactions take place into account. Context awareness also differentiates ubiquitous computing systems from the more traditional systems we have been discussing before, and is described by Dey and Abowd [2000] as "any information that can be used to characterize the situation of entities (i.e., whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves." In practice, context is often characterized by location, identity, time, and activity: the *where*, *who*, *when*, and *what*. A system will need to have the necessary (sensory) input to determine one or several of these context types.

What is important from a distributed-systems perspective, is that raw data as collected by various sensors is lifted to a level of abstraction that can be used by applications. A concrete example is detecting where a person is, for example in terms of GPS coordinates, and subsequently mapping that information to an actual location, such as the corner of a street, or a specific shop or other known facility. The question is where this processing of sensory input takes place: is all data collected at a central server connected to a database with detailed information on a city, or is it the user's smartphone where the mapping is done? Clearly, there are trade-offs to be considered.

Dey [2010] discusses more general approaches toward building context-aware applications. When it comes to combining flexibility and potential distribution, so-called **shared data spaces** in which processes are decoupled in time and space are attractive, yet as we shall see in later chapters, suffer from scalability problems. A survey on context-awareness and its relation to middleware and distributed systems is provided by Baldauf et al. [2007].

Ad. 4: Autonomy. An important aspect of most ubiquitous computing systems is that explicit systems management has been reduced to a minimum. In a ubiquitous computing environment there is simply no room for a systems administrator to keep everything up and running. As a consequence, the system as a whole should be able to act autonomously, and automatically react to changes. This requires a myriad of techniques of which several will be discussed throughout this book. To give a few simple examples, think of the following:

Address allocation: In order for networked devices to communicate, they need an IP address. Addresses can be allocated automatically using protocols like the Dynamic Host Configuration Protocol (DHCP) [Droms, 1997] (which requires a server) or Zeroconf [Guttman, 2001].

Adding devices: It should be easy to add devices to an existing system. A step towards automatic configuration is realized by the **Universal Plug and Play protocol** (**UPnP**) [UPnP Forum, 2008]. Using UPnP, devices can discover each other and make sure that they can set up communication channels between them.

Automatic updates: Many devices in a ubiquitous computing system should be able to regularly check through the Internet if their software should be updated. If so, they can download new versions of their components and ideally continue where they left off.

Admittedly, these are very simple examples, but the picture should be clear that manual intervention is to be kept to a minimum. We will be discussing many techniques related to self-management in detail throughout the book.

Ad. 5: Intelligence. Finally, Poslad [2009] mentions that ubiquitous computing systems often use methods and techniques from the field of artificial intelligence. What this means, is that in many cases a wide range of advanced algorithms and models need to be deployed to handle incomplete input, quickly react to a changing environment, handle unexpected events, and so on. The extent to which this can or should be done in a distributed fashion is crucial from the perspective of distributed systems. Unfortunately, distributed solutions for many problems in the field of artificial intelligence are yet to be found, meaning that there may be a natural tension between

the first requirement of networked and distributed devices, and advanced distributed information processing.

Mobile computing systems

As mentioned, mobility often forms an important component of pervasive systems, and many, if not all aspects that we have just discussed also apply to **mobile computing**. There are several issues that set mobile computing aside to pervasive systems in general (see also Adelstein et al. [2005] and Tarkoma and Kangasharju [2009]).

First, the devices that form part of a (distributed) mobile system may vary widely. Typically, mobile computing is now done with devices such as smartphones and tablet computers. However, completely different types of devices are now using the Internet Protocol (IP) to communicate, placing mobile computing in a different perspective. Such devices include remote controls, pagers, active badges, car equipment, various GPS-enabled devices, and so on. A characteristic feature of all these devices is that they use wireless communication. Mobile implies wireless so it seems (although there are exceptions to the rules).

Second, in mobile computing the location of a device is assumed to change over time. A changing location has its effects on many issues. For example, if the location of a device changes regularly, so will perhaps the services that are locally available. As a consequence, we may need to pay special attention to dynamically discovering services, but also letting services announce their presence. In a similar vein, we often also want to know where a device actually is. This may mean that we need to know the actual geographical coordinates of a device such as in tracking and tracing applications, but it may also require that we are able to simply detect its network position (as in mobile IP [Perkins, 2010; Perkins et al., 2011].

Changing locations also has a profound effect on communication. To illustrate, consider a (wireless) **mobile ad hoc network**, generally abbreviated as a **MANET**. Suppose that two devices in a MANET have discovered each other in the sense that they know each other's network address. How do we route messages between the two? Static routes are generally not sustainable as nodes along the routing path can easily move out of their neighbor's range, invalidating the path. For large MANETs, using a priori set-up paths is not a viable option. What we are dealing with here are so-called **disruption-tolerant networks**: networks in which connectivity between two nodes can simply not be guaranteed. Getting a message from one node to another may then be problematic, to say the least.

The trick in such cases, is not to attempt to set up a communication path from the source to the destination, but to rely on two principles. First, as we will discuss in Section 4.4, using special flooding-based techniques will allow a message to gradually spread through a part of the network, to eventually

reach the destination. Obviously, any type of flooding will impose redundant communication, but this may be the price we have to pay. Second, in a disruption-tolerant network, we let an intermediate node store a received message until it encounters another node to which it can pass it on. In other words, a node becomes a temporary carrier of a message, as sketched in Figure 1.14. Eventually, the message should reach its destination.

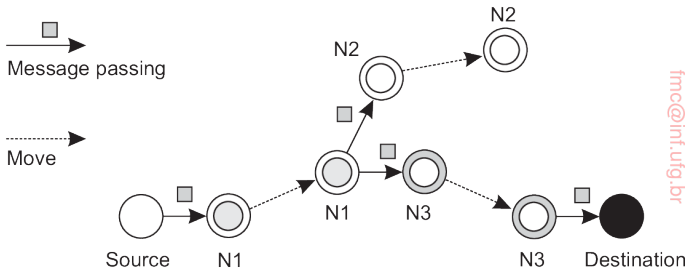

Figure 1.14: Passing messages in a (mobile) disruption-tolerant network.

It is not difficult to imagine that selectively passing messages to encountered nodes may help to ensure efficient delivery. For example, if nodes are known to belong to a certain class, and the source and destination belong to the same class, we may decide to pass messages only among nodes in that class. Likewise, it may prove efficient to pass messages only to well-connected nodes, that is, nodes who have been in range of many other nodes in the recent past. An overview is provided by Spyropoulos et al. [2010].

Note 1.11 (Advanced: Social networks and mobility patterns)
Not surprisingly, mobile computing is tightly coupled to the whereabouts of human beings. With the increasing interest in complex social networks [Vega-Redondo, 2007; Jackson, 2008] and the explosion of the use of smartphones, several groups are seeking to combine analysis of social behavior and information dissemination in so-called **pocket-switched networks** [Hui et al., 2005]. The latter are networks in which nodes are formed by people (or actually, their mobile devices), and links are formed when two people encounter each other, allowing their devices to exchange data.

The basic idea is to let information be spread using the ad hoc communications between people. In doing so, it becomes important to understand the structure of a social group. One of the first to examine how social awareness can be exploited in mobile networks were Miklas et al. [2007]. In their approach, based on traces on encounters between people, two people are characterized as either friends or strangers. Friends interact frequently, where the number of recurring encounters between strangers is low. The goal is to make sure that a message from Alice to Bob is eventually delivered.

As it turns out, when Alice adopts a strategy by which she hands out the message to each of her friends, and that each of those friends passes the message to Bob as soon as he is encountered, can ensure that the message reaches Bob with a delay exceeding approximately 10% of the best-attainable delay. Any other strategy, like forwarding the message to only 1 or 2 friends, performs much worse. Passing a message to a stranger has no significant effect. In other words, it makes a huge difference if nodes take friend relationships into account, but even then it is still necessary to judiciously adopt a forwarding strategy.

For large groups of people, more sophisticated approaches are needed. In the first place, it may happen that messages need to be sent between people in different *communities*. What do we mean by a community? If we consider a social network (where a vertex represents a person, and a link the fact that two people have a social relation), then a community is roughly speaking a group of vertices in which there are many links between its members and only few links with vertices in other groups [Newman, 2010]. Unfortunately, many community-detection algorithms require complete information on the social structure, making them practically infeasible for optimizing communication in mobile networks.

Hui et al. [2007] propose a number of decentralized community detection algorithms. In essence, these algorithms rely on letting a node i (1) detect the set of nodes it regularly encounters, called its familiar set F_i, and (2) incrementally expand its local community C_i, with $F_i \subseteq C_i$. Initially, C_i as well as F_i will be empty, but gradually, F_i will grow, and with it, C_i. In the simplest case, a node j is added to a community C_i as follows:

$$\text{Node i adds j to } C_i \text{ when } \frac{|F_j \cap C_i|}{|F_j|} > \lambda \text{ for some } \lambda > 0$$

In other words, when the fraction of j's familiar set substantially overlaps with the community of i, then node i should add j to its community. Also, we have the following for merging communities:

$$\text{Merge two communities when } |C_i \cap C_j| > \gamma |C_i \cup C_j| \text{ for some } \gamma > 0$$

which means that two communities should be merged when they have a significant number of members in common. (In their experiments, Hui et al. found that setting $\lambda = \gamma = 0.6$ lead to good results.)

Knowing communities, in combination with the connectivity of a node in either a community, or globally, can subsequently be used to efficiently forward messages in a disruption-tolerant network, as explained by Hui et al. [2011].

Obviously, much of the performance of a mobile computing system depends on how nodes move. In particular, in order to pre-assess the effectiveness of new protocols or algorithms, having an idea on which mobility patterns are actually realistic is important. For long, there was not much data on such patterns, but recent experiments have changed that.

Various groups have started to collect statistics on human mobility, of which the traces are used to drive simulations. In addition, traces have been used to derive more realistic mobility models (see, e.g., Kim et al. [2006b]). However,

understanding human mobility patterns in general remains a difficult problem. González et al. [2008] report on modeling efforts based on data collected from 100,000 cell-phone users during a six-month period. They observed that the displacement behavior could be represented by the following, relatively simple distribution:

$$\mathbb{P}[\Delta r] = (\Delta r + \Delta r_0)^{-\beta} \cdot e^{-\Delta r/\kappa}$$

in which Δr is the actual displacement and $\Delta r_0 = 1.5$km a constant initial displacement. With $\beta = 1.75$ and $\kappa = 400$, this leads to the distribution shown in Figure 1.15.

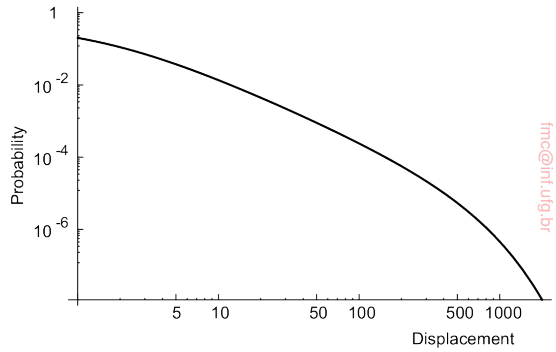

Figure 1.15: The distribution of displacement of (mobile) cell-phone users.

We can conclude that people tend to stay put. In fact, further analysis revealed that people tend to return to the same place after 24, 48, or 72 hours, clearly showing that people tend to go the same places. In a follow-up study, Song et al. [2010] could indeed show that human mobility is actually remarkably well predictable.

Sensor networks

Our last example of pervasive systems is **sensor networks**. These networks in many cases form part of the enabling technology for pervasiveness and we see that many solutions for sensor networks return in pervasive applications. What makes sensor networks interesting from a distributed system's perspective is that they are more than just a collection of input devices. Instead, as we shall see, sensor nodes often collaborate to efficiently process the sensed data in an application-specific manner, making them very different from, for example, traditional computer networks. Akyildiz et al. [2002] and Akyildiz et al. [2005] provide an overview from a networking perspective. A more systems-oriented introduction to sensor networks is given by Zhao and Guibas [2004], but also Karl and Willig [2005] will show to be useful.

A sensor network generally consists of tens to hundreds or thousands of relatively small nodes, each equipped with one or more sensing devices. In

addition, nodes can often act as actuators [Akyildiz and Kasimoglu, 2004], a typical example being the automatic activation of sprinklers when a fire has been detected. Many sensor networks use wireless communication, and the nodes are often battery powered. Their limited resources, restricted communication capabilities, and constrained power consumption demand that efficiency is high on the list of design criteria.

When zooming into an individual node, we see that, conceptually, they do not differ a lot from "normal" computers: above the hardware there is a software layer akin to what traditional operating systems offer, including low-level network access, access to sensors and actuators, memory management, and so on. Normally, support for specific services is included, such as localization, local storage (think of additional flash devices), and convenient communication facilities such as messaging and routing. However, similar to other networked computer systems, additional support is needed to effectively deploy sensor network *applications*. In distributed systems, this takes the form of middleware. For sensor networks, instead of looking at middleware, it is better to see what kind of programming support is provided, which has been extensively surveyed by Mottola and Picco [2011].

One typical aspect in programming support is the scope provided by communication primitives. This scope can vary between addressing the physical neighborhood of a node, and providing primitives for systemwide communication. In addition, it may also be possible to address a specific group of nodes. Likewise, computations may be restricted to an individual node, a group of nodes, or affect all nodes. To illustrate, Welsh and Mainland [2004] use so-called **abstract regions** allowing a node to identify a neighborhood from where it can, for example, gather information:

```
1  region  = k_nearest_region.create(8);
2  reading = get_sensor_reading();
3  region.putvar(reading_key, reading);
4  max_id  = region.reduce(OP_MAXID, reading_key);
```

In line 1, a node first creates a region of its eight nearest neighbors, after which it fetches a value from its sensor(s). This reading is subsequently written to the previously defined region to be defined using the key `reading_key`. In line 4, the node checks whose sensor reading in the defined region was the largest, which is returned in the variable `max_id`.

As another related example, consider a sensor network as implementing a distributed database, which is, according to Mottola and Picco [2011], one of four possible ways of accessing data. This database view is quite common and easy to understand when realizing that many sensor networks are deployed for measurement and surveillance applications [Bonnet et al., 2002]. In these cases, an operator would like to extract information from (a part of) the network by simply issuing queries such as "What is the northbound traffic load on highway 1 as Santa Cruz?" Such queries resemble those of traditional

databases. In this case, the answer will probably need to be provided through collaboration of many sensors along highway 1, while leaving other sensors untouched.

To organize a sensor network as a distributed database, there are essentially two extremes, as shown in Figure 1.16. First, sensors do not cooperate but simply send their data to a centralized database located at the operator's site. The other extreme is to forward queries to relevant sensors and to let each compute an answer, requiring the operator to aggregate the responses.

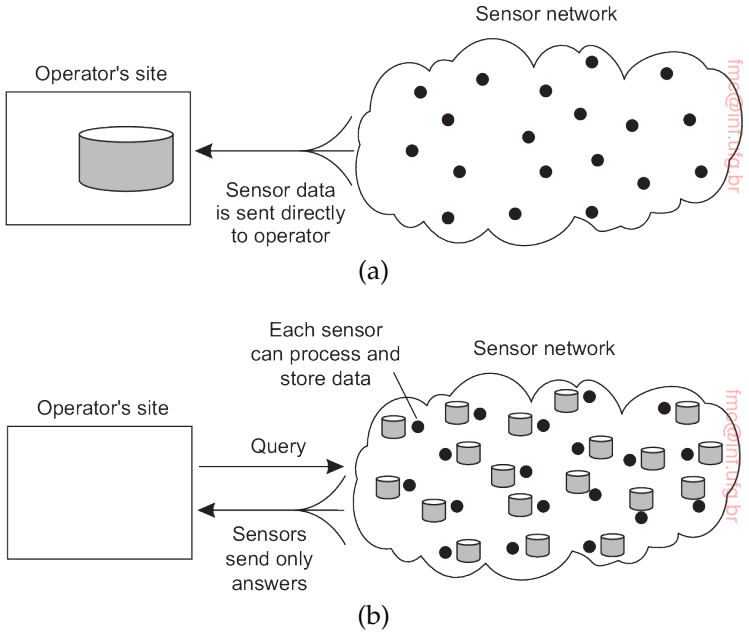

Figure 1.16: Organizing a sensor network database, while storing and processing data (a) only at the operator's site or (b) only at the sensors.

Neither of these solutions is very attractive. The first one requires that sensors send all their measured data through the network, which may waste network resources and energy. The second solution may also be wasteful as it discards the aggregation capabilities of sensors which would allow much less data to be returned to the operator. What is needed are facilities for **in-network data processing**, similar to the previous example of abstract regions.

In-network processing can be done in numerous ways. One obvious one is to forward a query to all sensor nodes along a tree encompassing all nodes and to subsequently aggregate the results as they are propagated back to the root, where the initiator is located. Aggregation will take place where two or more branches of the tree come together. As simple as this scheme may

sound, it introduces difficult questions:

- How do we (dynamically) set up an efficient tree in a sensor network?
- How does aggregation of results take place? Can it be controlled?
- What happens when network links fail?

These questions have been partly addressed in TinyDB, which implements a declarative (database) interface to wireless sensor networks [Madden et al., 2005]. In essence, TinyDB can use any tree-based routing algorithm. An intermediate node will collect and aggregate the results from its children, along with its own findings, and send that toward the root. To make matters efficient, queries span a period of time allowing for careful scheduling of operations so that network resources and energy are optimally consumed.

However, when queries can be initiated from different points in the network, using single-rooted trees such as in TinyDB may not be efficient enough. As an alternative, sensor networks may be equipped with special nodes where results are forwarded to, as well as the queries related to those results. To give a simple example, queries and results related to temperature readings may be collected at a different location than those related to humidity measurements. This approach corresponds directly to the notion of publish/subscribe systems.

Note 1.12 (Advanced: When energy starts to become critical)
As mentioned, many sensor networks need to operate on an energy budget coming from the use of batteries or other limited power supplies. An approach to reduce energy consumption, is to let nodes be active only part of the time. More specifically, assume that a node is repeatedly active during T_{active} time units, and between these active periods, it is suspended for $T_{\text{suspended}}$ units. The fraction of time that a node is active is known as its **duty cycle** τ, that is,

$$\tau = \frac{T_{\text{active}}}{T_{\text{active}} + T_{\text{suspended}}}$$

Values for τ are typically in the order of $10 - 30\%$, but when a network needs to stay operational for periods exceeding many months, or even years, attaining values as low as 1% are critical.

A problem with duty-cycled networks is that, in principle, nodes need to be active at the same time for otherwise communication would simply not be possible. Considering that while a node is suspended, only its local clock continues ticking, and that these clocks are subject to drifts, waking up at the same time may be problematic. This is particularly true for networks with very low duty cycles.

When a group of nodes are active at the same time, the nodes are said to form a **synchronized group**. There are essentially two problems that need to be addressed. First, we need to make sure that the nodes in a synchronized group remain active at the same time. In practice, this turns out to be relatively simple

if each node communicates information on its current local time. Then, simple local clock adjustments will do the trick. The second problem is more difficult, namely how two different synchronized groups can be merged into one in which all nodes are synchronized. Let us take a closer look at what we are facing. Most of the following discussion is based on material by Voulgaris et al. [2016].

In order to have two groups be merged, we need to first ensure that one group detects the other. Indeed, if their respective active periods are completely disjoint, there is no hope that any node in one group can pick up a message from a node in the other group. In an *active detection method*, a node will send a *join message* during its suspended period. In other words, while it is suspended, it temporarily wakes up to elicit nodes in other groups to join. How big is the chance that another node will pick up this message? Realize that we need to consider only the case when $\tau < 0.5$, for otherwise two active periods will always overlap, meaning that two groups can easily detect each other's presence. The probability P_{da} that a join message can be picked up during another node's active period, is equal to

$$P_{da} = \frac{T_{\text{active}}}{T_{\text{suspended}}} = \frac{\tau}{1-\tau}$$

This means that for low values of τ, P_{da} is also very small.

In a *passive detection method*, a node skips the suspended state with (a very low) probability P_{dp}, that is, it simply stays active during the $T_{\text{suspended}}$ time units following its active period. During this time, it will be able to pick up any messages sent by its neighbors, who are, by definition, member of a different synchronized group. Experiments show that passive detection is inferior to active detection.

Simply stating that two synchronized groups need to merge is not enough: if A and B have discovered each other, which group will adapt the duty-cycle settings of the other? A simple solution is to use a notion of cluster IDs. Each node starts with a randomly chosen ID and effectively also a synchronized group having only itself as member. After detecting another group B, all nodes in group A join B if and only if the cluster ID of B is larger than that of A.

Synchronization can be improved considerably using so-called *targeted join messages*. Whenever a node N receives a join message from a group A with a lower cluster ID, it should obviously not join A. However, as N now knows when the active period of A is, it can send a join message exactly during that period. Obviously, the chance that a node from A will receive that message is very high, allowing the nodes from A to join N's group. In addition, when a node decides to join another group, it can send a special message to its group members, giving the opportunity to quickly join as well.

Figure 1.17 shows how quickly synchronized groups can merge using two different strategies. The experiments are based on a 4000-node mobile network using realistic mobility patterns. Nodes have a duty cycle of less than 1%. These experiments show that bringing even a large mobile, duty-cycled network to a state in which all nodes are active at the same time is quite feasible. For further information, see Voulgaris et al. [2016].

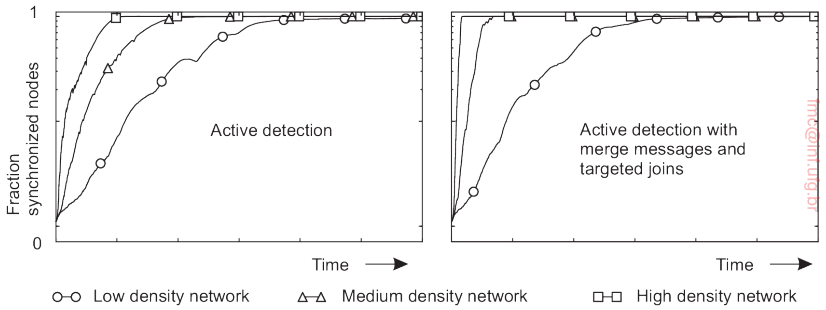

Figure 1.17: The speed by which different synchronized groups can merge.

1.4 Summary

Distributed systems consist of autonomous computers that work together to give the appearance of a single coherent system. This combination of independent, yet coherent collective behavior is achieved by collecting application-independent protocols into what is known as middleware: a software layer logically placed between operating systems and distributed applications. Protocols include those for communication, transactions, service composition, and perhaps most important, reliability.

Design goals for distributed systems include sharing resources and ensuring openness. In addition, designers aim at hiding many of the intricacies related to the distribution of processes, data, and control. However, this distribution transparency not only comes at a performance price, in practical situations it can never be fully achieved. The fact that trade-offs need to be made between achieving various forms of distribution transparency is inherent to the design of distributed systems, and can easily complicate their understanding. One specific difficult design goal that does not always blend well with achieving distribution transparency is scalability. This is particularly true for geographical scalability, in which case hiding latencies and bandwidth restrictions can turn out to be difficult. Likewise, administrative scalability by which a system is designed to span multiple administrative domains, may easily conflict goals for achieving distribution transparency.

Matters are further complicated by the fact that many developers initially make assumptions about the underlying network that are fundamentally wrong. Later, when assumptions are dropped, it may turn out to be difficult to mask unwanted behavior. A typical example is assuming that network latency is not significant. Other pitfalls include assuming that the network is reliable, static, secure, and homogeneous.

Different types of distributed systems exist which can be classified as being oriented toward supporting computations, information processing, and pervasiveness. Distributed computing systems are typically deployed for high-performance applications often originating from the field of parallel computing. A field that emerged from parallel processing was initially grid computing with a strong focus on worldwide sharing of resources, in turn leading to what is now known as cloud computing. Cloud computing goes beyond high-performance computing and also supports distributed systems found in traditional office environments where we see databases playing an important role. Typically, transaction processing systems are deployed in these environments. Finally, an emerging class of distributed systems is where components are small, the system is composed in an ad hoc fashion, but most of all is no longer managed through a system administrator. This last class is typically represented by pervasive computing environments, including mobile-computing systems as well as sensor-rich environments.

