CLOUD COMPUTING TECHNOLOGIES AND DISTRIBUTED SYSTEMS

PROFESSOR USTIJANA RECHKOSKA-SHIKOSKA, PH.D.

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 networks** 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. 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 computers** are 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.

WHAT IS A DISTRIBUTED SYSTEM?

- 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.

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.

CHARACTERISTIC 2

Single coherent system

As mentioned, a distributed system should appear as a single coherent system. 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.

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.

IMPORTANCE OF MIDDLEWARE

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**

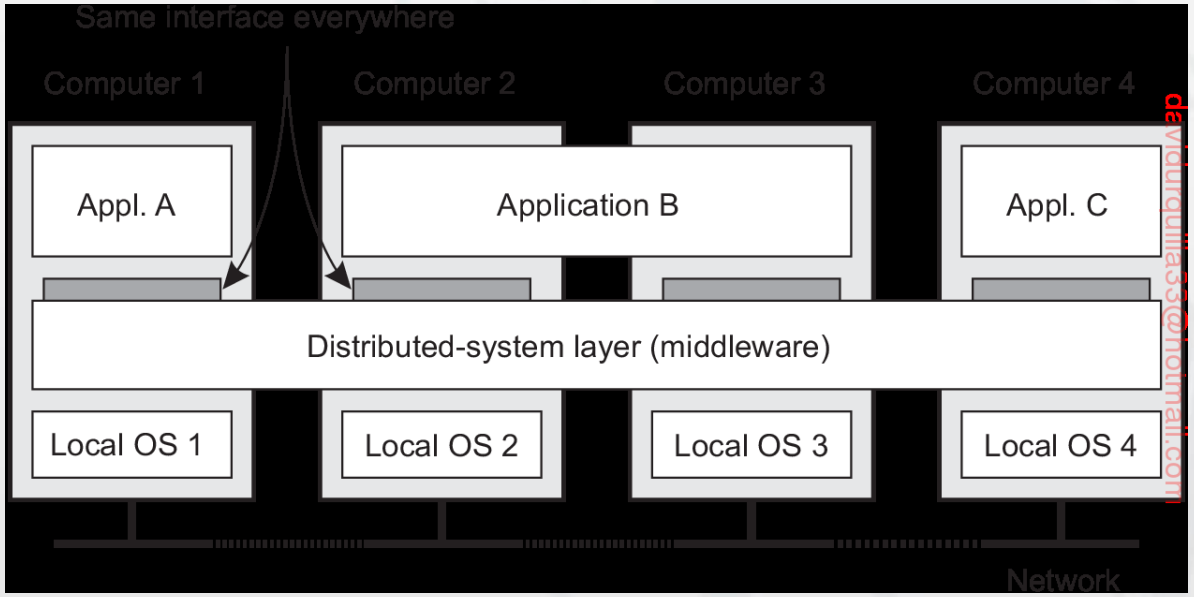

Figure 1.1 shows four networked computers and three applications, of which application B is distributed across computers 2 and 3. Each application is offered the same interface. The distributed system provides the means for components of a single distributed application to communicate with each other, but also to let different applications communicate

WHAT DOES MIDDLEWARE DO?

Communication: A common communication service is the so-called **Remote Procedure Call** (**RPC**). 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**.

Service composition: It is becoming increasingly common to develop new applications by taking existing programs and gluing them together. 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. .

Reliability: As a last example, there has been a wealth of research on providing enhanced functions for building reliable distributed applications.

FOUR IMPORTANT DESIGN GOALS

1- **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. Connecting users and resources also makes it easier to collaborate and exchange information, as is illustrated by the success of the Internet.

2-**Making distribution transparent:** make the distribution of processes and resources **transparent**, that is, invisible, to end users and applications.

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

3-Being open: 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.

4-Being scalable:scalability has become one of the most important design goals for developers of distributed systems. Three dimensions for scalability: size,geographical and administrative.

SUPPORTING RESOURCE SHARING

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.

DISTRIBUTION TRANSPARENCY

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.

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.

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

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.

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.

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 differentparameters that need to be considered: storage,exemption,sharing,refreshing.

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 can be measure at three dimensions: size,geographical, and administrative.

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.

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.

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.

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.

Administrative scalability:

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

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.

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.

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.

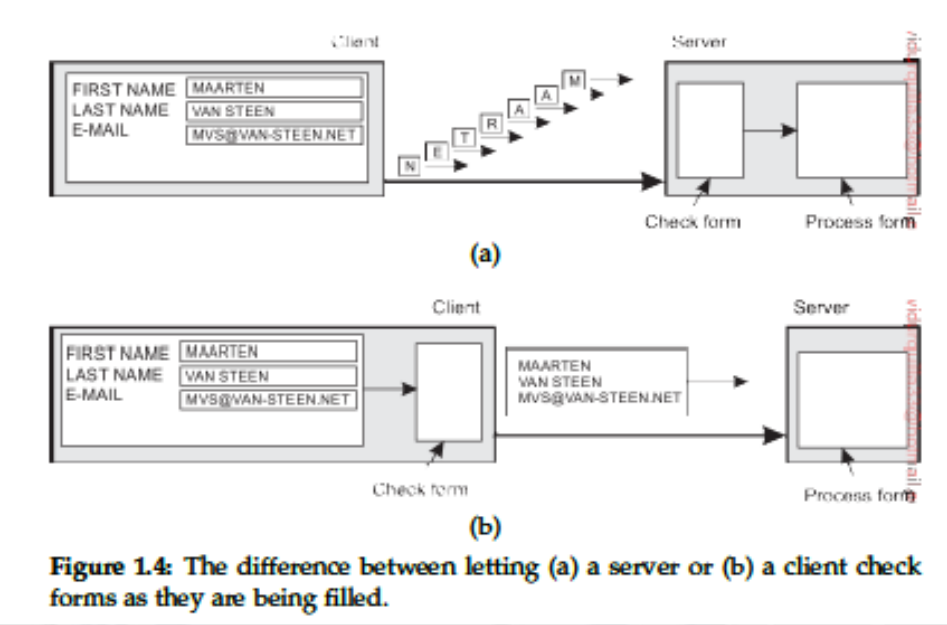

Figure 1.4: The difference between letting (a) a server or (b) a client check forms as they are being filled.

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 its.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, its.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.

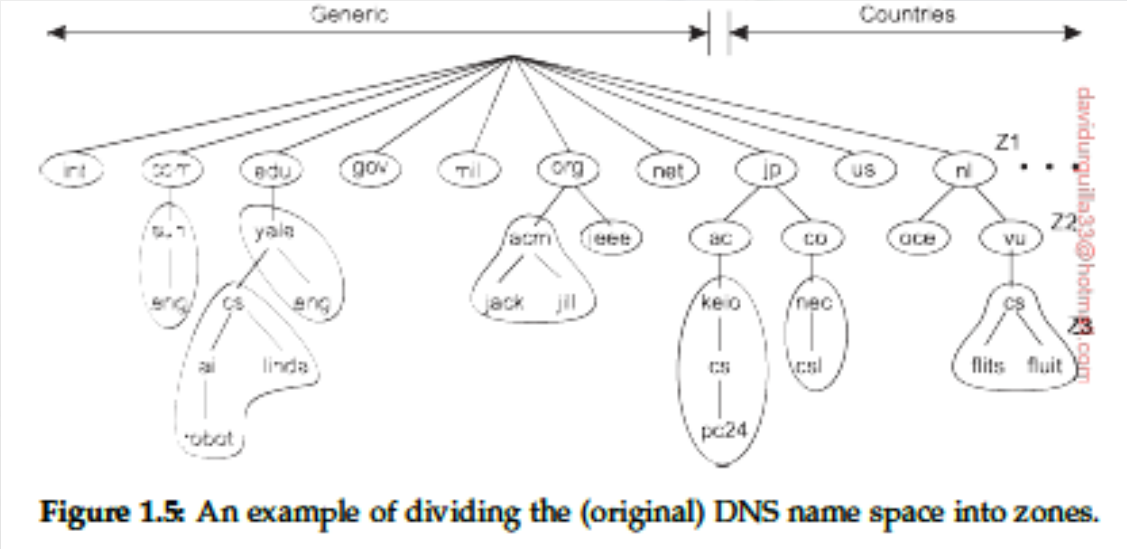

Figure 1.5: An example of dividing the (original) DNS name space into zones.

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.

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. 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.

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 everyone makes 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 DIFFERENT TYPES OF DISTRIBUTED SYSTEMS

We make a distinction between distributed computing systems, distributed information systems, and pervasive systems (which are naturally distributed).

1-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.

CLUSTER COMPUTING

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.

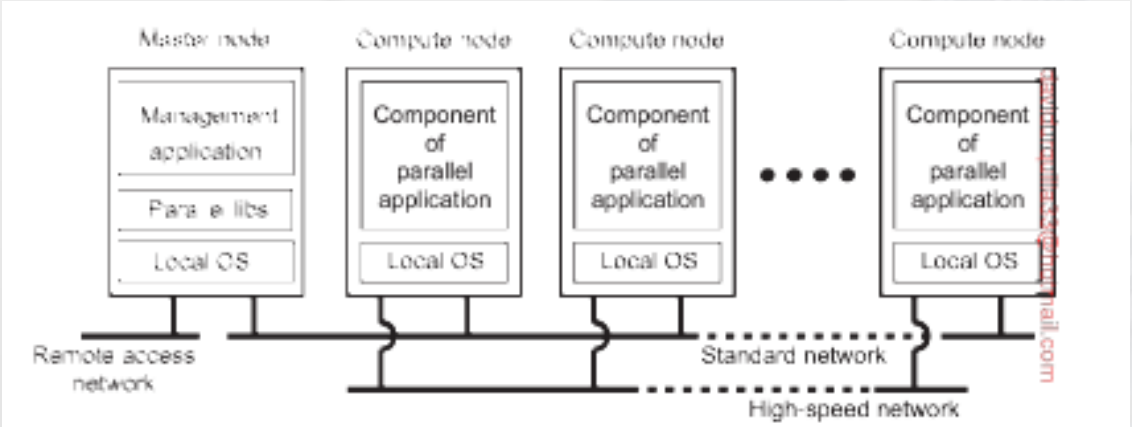

Figure 1.7: An example of a cluster computing system.

GRID COMPUTING

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). 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.

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.

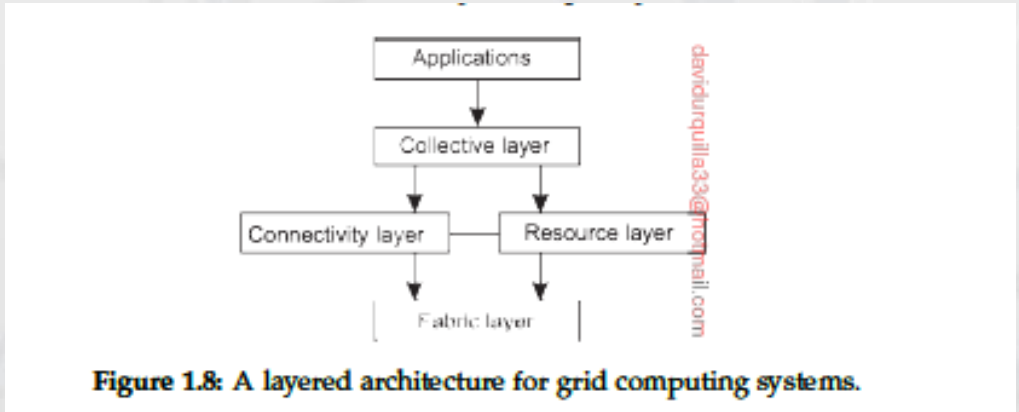

Figure 1.8: A layered architecture for grid computing systems.

CLOUD COMPUTING

The clouds are organized in four layers

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 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. Also like operating systems, the platform layer provides higher-level abstractions for storage and such.

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.

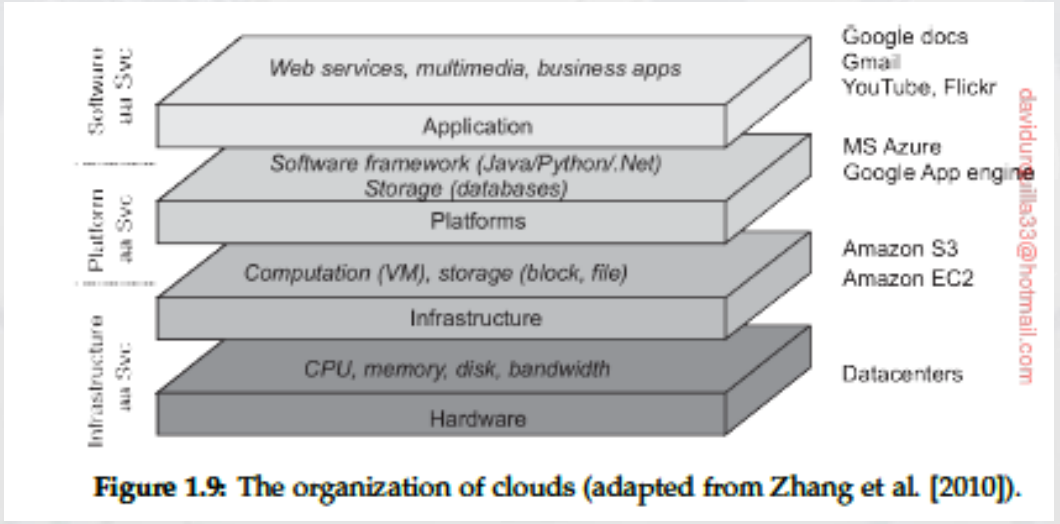

Figure 1.9: The organization of clouds (adapted from Zhang et al. [2010]).

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.

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 lead to a huge industry that concentrates on **Enterprise Application Integration** (**EAI**).

DISTRIBUTION TRANSACTION PROCESSING

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**s), that is, procedure calls to remote servers, are often also encapsulated in a transaction, leading to what is known as a **transactional RPC**.

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.

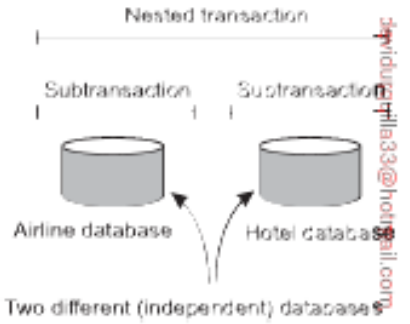

Figure 1.11: A nested transaction.

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. 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 **messageoriented 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.

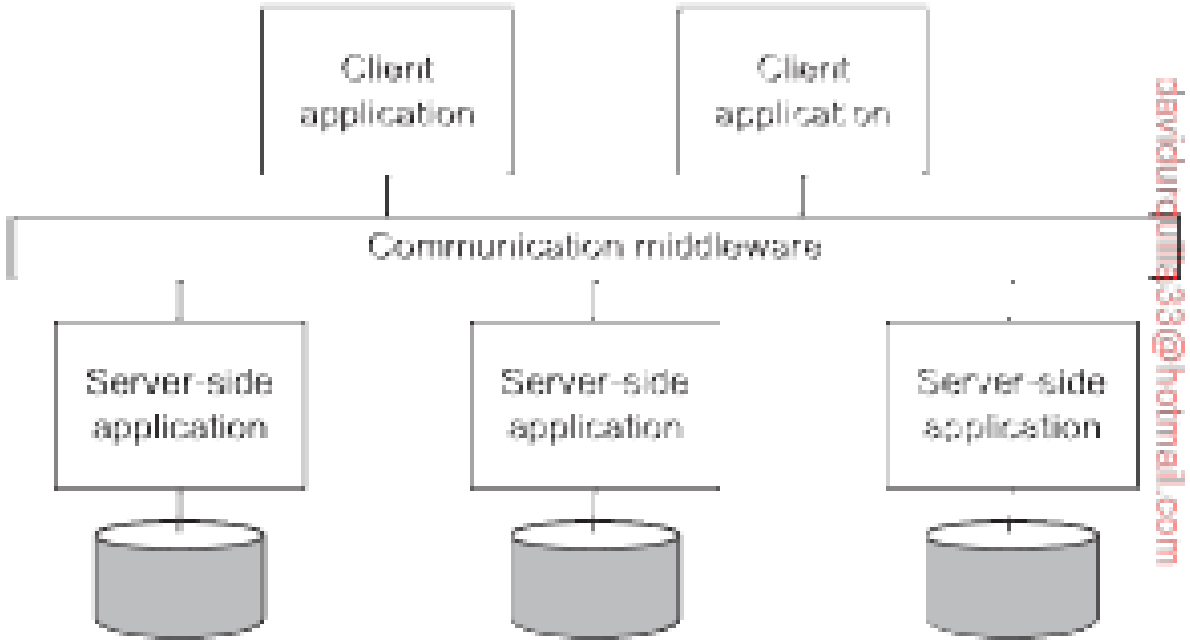

Figure 1.13: Middleware as a communication facilitator in enterprise application integration.

PERVASIVE SYSTEMS

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. 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.

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.

UBIQUITOUS COMPUTING SYSTEMS

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.

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.

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

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. Some examples of this:

- **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
- **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.**
- **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.

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

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. 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.

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. 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 ntolerant 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.

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. 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.

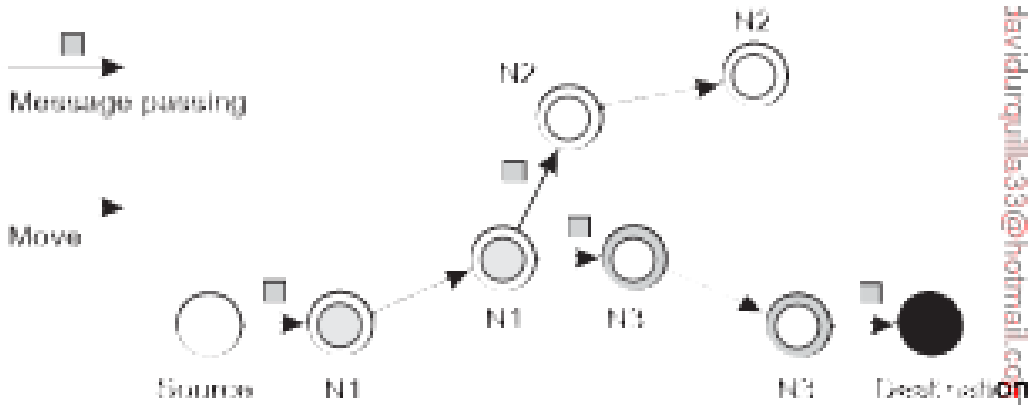

Figure 1.14: Passing messages in a (mobile) disruption-tolerant network.

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.
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 , a typical example being the automatic activation of sprinklers when a fire has been detected.

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

Consider a sensor network as implementing a distributed database, which is 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. 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. 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

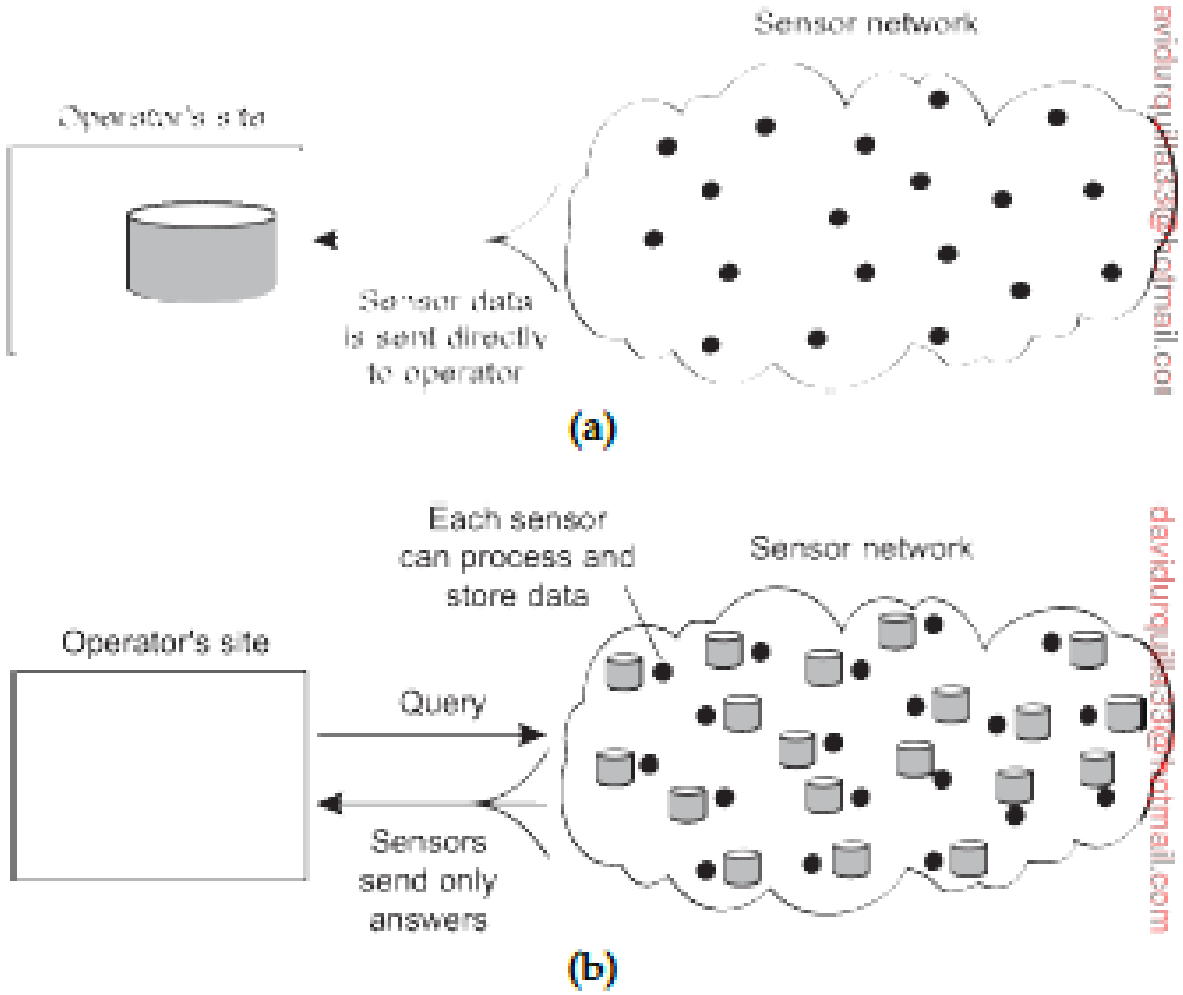

Figure 1.16: Organizing a sensor network database, while storing and processing data (a) only at the operator's site or (b) only at the sensors.

WHAT DID WE LEARN?

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 applicationindependent 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.