CHAPTER

THE ARCHITECTURE VIEW

5

CHAPTER CONTENTS

5.1 RATIONALE AND OVERVIEW

A distributed system by definition comprises at least two components, but some systems comprise many components and exhibit very complex behavior as a result. Architecture describes the way in which systems are structured and how the constituent components are interconnected and the relationships between the components; this includes the organization of communication and control channels between pairs or groups of components. The choice of architecture for a given system will impact on its scalability, robustness, and performance, as well as the efficiency with which the resources of the system are used; in fact, it potentially impacts all aspects of the system's effectiveness.

There are many reasons why systems are designed as a number of components, some relating to functionality, some relating to access to resources, and some relating to scale of systems, among others. The reasons why systems are built as a collection of components, and also the wide variety of system architectures that arise as a result, are discussed. There are a number of commonly occurring structures and patterns and also many application-specific structures. In addition, some structures are static, that is, the connectivity between components is decided at design time, while other systems have dynamic structures and connectivity, which arises due to the operating context and the state of the wider system itself in which the application runs. Mechanisms to support dynamic component discovery and connectivity are explored, as well as techniques to automatically configure services and allocate roles to server instances.

The effects of heterogeneity in systems are examined, as well as ways in which the heterogeneity challenge can be overcome with services such as middleware and techniques such as hardware virtualization. Structure at the component level is examined with the aid of practical examples of refactoring and the creation of a software library. The use of replication as an architectural design technique to meet nonfunctional requirements including robustness, availability, and responsiveness is explored in an extensive practical activity.

5.2 THE ARCHITECTURE VIEW

The architecture of a system is its structure. Distributed applications comprise several components that have communication relationships and control relationships with other components. These components may be organized into a hierarchical structure where components occupy different layers depending on their role.

Perhaps, the single largest influence on the overall quality and effectiveness of a distributed application is its architecture. The way in which an application is divided into components, and the way

these components are subsequently organized into a suitable structure, supporting communication and control has a very strong impact on the performance achieved. Performance characteristics influenced by architecture include scalability (e.g., in terms of the way in which the performance is affected by increases in the number of components in the system or by increases in throughput measured as the number of transactions per unit time), flexibility (e.g., in terms of the coupling between components and the extent that dynamic (re)configuration is possible), and efficiency (e.g., in terms of the communication intensity between components measured by the number of messages sent and the overall communication overhead incurred).

There is an important difference between a physical architecture and a logical architecture. Physical architecture describes the configuration of computers and their interconnectivity. Developers of distributed applications and systems are primarily concerned with the logical architecture in which the logical connectivity of components is important, but the physical location of these components (i.e., their mapping to actual computers) is not a concern. The application logic is considered distributed even if all the processes that carry out the work reside on the same physical computer. It is common that processes are actually spread across multiple computers, and this introduces not only several special considerations, most obviously the communication aspect, but also timing and synchronization issues.

5.2.1 SEPARATION OF CONCERNS

An application is defined and distinguished by its functionality, that is, what it actually does. In achieving this functionality, there are a number of different concerns at the business logic level. These concerns may include specific aspects of functionality, meeting nonfunctional requirements such as robustness or scalability, as well as other issues such as accessibility and security. These various requirements and behaviors are mapped onto the actual components in ways that are highly application-dependent, taking into account not only the functionalities themselves but the specific prioritization among the functionalities that is itself application-dependent.

It should be possible to identify the specific software component(s) that provides a particular functionality. This is because a functional requirement is something the system must actually do, or perform, and thus can usually be explicitly expressed in design documentation and eventually translated into code. For example, encryption of a message prior to sending may be performed in a code function called *EncryptMessage*(), which is contained within a specific component of the application. However, it is not generally possible to implement nonfunctional requirements directly in code (by implication of their "nonfunctional" nature). Take a couple of very common requirements such as scalability and efficiency; almost all distributed applications have these among their nonfunctional requirements, but even the most experienced software developers will be unable to write functions *Scalability*() or *Efficiency*() to provide these characteristics. This is because scalability and efficiency are *not* functions as such they are qualities. Instead of providing a clearly demarked function, the entire application (or certain key parts of it) must be designed to ensure that the overall resulting structure and behavior are scalable and efficient. There may however be functions that directly or indirectly contribute to the achievement of nonfunctional requirements. For example, a method such as *CompressMessageBeforeSending*() may contribute to scalability as well as efficiency. However, the achievement of scalability and efficiency additionally depend on the higher-level structure such

as the way in which the components are themselves coupled together, as well as specific aspects of behavior.

One of the key steps in defining the architecture of an application is the separation of the concerns, that is, deciding how to split the application logic so that it can be spread across the various components. This must be done very carefully. Architectures where the functional separation of the business logic across the components is quite clear with well-defined boundaries tend to be easier to implement and are potentially more scalable. Situations where there are many components but the functionality is not clearly split across them are more likely to suffer performance or efficiency problems (due to additional communication requirements) and also are likely to be less robust because of complex dependencies between components and ongoing updates through the system's life cycle; this is because when components are tightly coupled with one another, it is very difficult to upgrade one component without the possibility of destabilizing several others.

Often, services such as name services and broker services are employed specifically to decouple components to ensure the components themselves remain as simple and independent as possible; this promotes flexibility in terms of run-time configuration and agility in terms of through lifetime maintenance and upgrade.

5.2.2 NETWORKING AND DISTRIBUTION

Not all applications that use the network to communicate are classified as distributed applications.

A network application is one in which the network conveys messages between components but where the application logic is not spread across the components, for example, situations where one of the components is simply a user interface as with a terminal emulator, such as a Telnet client. The user runs a local application that provides an interface to another computer so that commands can be executed remotely and the results presented on the user's display. The user's application in this case is only an access portal or interface that allows the user to execute commands on the remote computer. It does not actually contain any application logic; rather, it connects to the remote computer and sends and receives messages in predefined ways. The application(s) that is used remains remote to the user.

In a network application, the two or more components are usually explicitly visible to users; for example, they may have to identify each component and connect to them (as in the Telnet example above, where the user is aware that they are logging into a remote site). A further example is when using a web browser to connect to a specific web page, the user is aware that the resource being accessed is remote to them (usually, the user has explicitly provided the web page URL or clicked on a hyperlink).

In contrast with network applications, the terms “distributed application” and “distributed system” imply that the logic and structure of an application are somehow distributed over multiple components, ideally in such a way that the users of the application are unaware of the distribution itself.

The main difference is thus transparency (in addition to the differentiation as to whether the business logic is distributed); the goal of good distributed system design is to create the illusion of a single entity and to hide the separation of components and the boundaries between physical computers. The extent of transparency is a main metric for measuring the quality of a distributed application or system. This issue is discussed in more detail later in the chapter.

5.2.3 COMPLEXITY IN DISTRIBUTED SYSTEMS

Managing complexity is a main concern in the design and development of distributed applications and systems. Systems can be highly complex in terms of their structure, their functionality, and/or their behavior.

There are a wide variety of sources of complexity; common categories include the following:

- The overall size of the system. This includes the number of computers, the number of software components, the amount of data, and the number of users.
- The extent of functionality. This includes the complexity of specific functions and also the variety and breadth of functionality across the system.
- The extent of interaction. This includes interaction between different features/functions, as well as communication between software components, the nature and extent of communication between users and the system, and possibly indirect interaction between users as a result of using the system (e.g., in applications such as banking with shared accounts and games).
- The speed at which the system operates. This includes the rate at which user requests arrive at the system, the throughput of the system in terms of the number of transactions completed in a given amount of time, and the amount of internal communication that occurs when processing transactions.
- Concurrency. This includes users submitting requests for service concurrently and also the effects arising from having many software components and also many processing elements operating in parallel within most systems.
- Reconfigurations. This includes forced reconfiguration caused by failure of various components, as well as purposeful reconfiguration (automatic or manual) to upgrade functionality or to improve efficiency.
- External or environmental conditions. This broad category of factors includes power failures that affect host computers and dynamic load levels in the communication network.

From this list, it is apparent that there are very many causes of complexity. A certain level of complexity is inevitable to achieve the necessary richness of behavior required for the applications to meet their design goals. It is not possible to entirely eliminate complexity.

Complexity is undesirable because it makes it difficult to understand all aspects of systems and their possible behaviors completely, and thus, it is more likely that weaknesses in terms of poor design or configuration can occur in more complex systems. Many systems are so complex that no individual person can understand the entire system. With such systems, it is very difficult to predict specific behavior in certain circumstances or to identify causes of faults or inefficiencies, and it is also very time-consuming to configure these systems for optimal behavior, if even possible in a realistic time frame.

Given that it is not possible to eliminate complexity from distributed applications, best practice is to reduce complexity where opportunities arise and to avoid introducing unnecessary complexity. This requires that designers consider the available options very carefully and make a special effort to understand the consequences of the various strategies and mechanisms they employ. The architecture of a system potentially has a large impact on its overall complexity because it describes the way in which components are connected together and the way that communication and control occur between these components.

5.2.4 LAYERED ARCHITECTURES

A flat architecture is where all of the components operate at the same level; there is no central coordination or control; instead, these systems can be described as collaborative or self-organizing. Such architectures occur in some natural systems in which large numbers of very simple organizations such as insects or cells in substances such as molds interact to achieve structures and/or behaviors beyond those capable of an individual element. This concept is called emergence and has been used effectively in some software systems such as agent-based systems.

However, such approaches rely on system characteristics that include having large numbers of similar entities as well as randomly occurring interactions between only neighboring entities, and these systems tend to work best when the entities themselves are simple in terms of the knowledge they hold and the functions they perform. Scalability is a serious challenge when the communication between components extends beyond immediate neighbors or where interactions occur at such a rate as to exceed the communication bandwidth available.

Most distributed computing systems contain much smaller numbers of components than occur in emergent systems (although there can still be large numbers running into the thousands). However, the various components are typically not identical across the system. There may be groups of identical components such as where a particular service comprises a number of replica server components to achieve robustness and/or performance, but such groups will be subsystems and effectively cogs in a larger machine. The communication and control requirements between components of distributed systems are not usually uniform; it is likely that some components coordinate or control the operation of others and also that some components interact intensely with some specific other components and much less, or not at all with others.

Layered software architectures comprise multiple layers of components that are placed into logical groupings based on the type of functionality they provide or based on their interactions with other components, such that interlayer communication occurs between adjacent layers. Applications and their subcomponents that interface directly with users occupy the upper layer of the architecture, services are lower down, the operating system then comes next, while components such as device drivers that interface with the system hardware are located at the bottom layers of the architecture. Layers can also be used to organize the components within a specific application or service; see Figure 5.1.

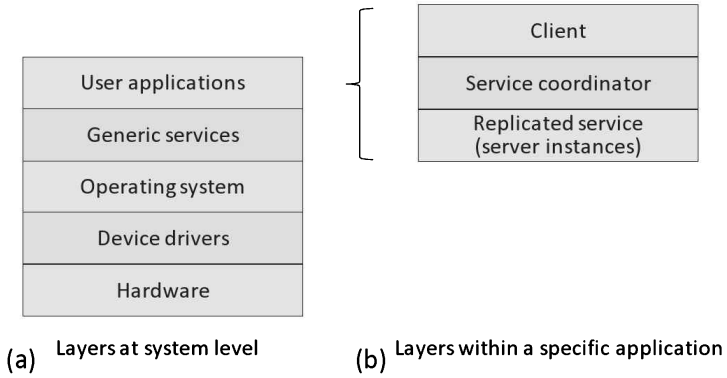


FIGURE 5.1

Generalization of the use of layers to provide structure.

Figure 5.1 illustrates in a generalized way how systems can be organized into layers to provide structure and thus manage complexity. Modern systems are too complex for users to be able to understand in their entirety, and therefore, it is difficult and cumbersome to make configuration choices across the full range of functionality in order to use the system. Separation into layers limits the scope of each layer and allows relevant configuration to occur while abstracting away details that are the concern of other layers. Part (a) of the figure shows how layers can be used to separate the concerns of applications from those of systems software and hardware. To put this into context, when using a particular application, the user should only have to interface with the application itself. It is undesirable from a usability viewpoint for the user to have to configure support services or make adjustments to operating settings in order to use the application. A simple example of this in practice is the use of a word processing application to write this paragraph. The application is one of many on my computer that can use the keyboard. The application indicates to the operating system that it requires input from the keyboard and the operating system performs the input stream mapping from the keyboard device driver to the word processing process without the user having to get involved or even being aware of this taking place. The operating system automatically maps the keyboard device to other processes if the user switches between different applications (e.g., once I get to the end of this paragraph, I might check my e-mail before continuing with the book). There is also the issue of hardware updates; if I replace my keyboard with a different one (perhaps one with additional function keys or an integrated roller ball), it is likely that a new device driver will be needed for the new keyboard. I do not want to have to reconfigure my word processing application to accommodate the new keyboard; the operating system should remap the input stream of my application process to the new device driver in a way that is transparent to the process itself and thus to the user.

Part (b) of the figure illustrates how applications can themselves be internally structured into several layers (this ability to subdivide also applies to the other parts of the system shown in part (a)). An important example is shown in which an application is divided into several components, of which the user is only exposed to the client part (hence, it is shown as the topmost component; conceptually, the user is looking down from above). This theme is explored in detail in Section 5.5 of this chapter.

The beneficial use of layers in network protocol stacks has been discussed in Chapter 3 and provides further justification of the value of layered structures to ensure maintainability and manageability of complex systems with rich functionality.

Layered architectures are very popular for distributed systems. The reasons for this include the following:

- Within a distributed system, there may be many distributed applications. These applications may each share some services; a good example of this is an authentication service that thus ensures that specific users have consistent access rights to services across the entire system and thus the security of the system is less likely to suffer weaknesses that would arise if some points of entry are less well protected than others. It can be useful to logically separate out the distributed end applications (the ones that provide service to users), from the service applications (which may themselves be distributed but provide services to other applications rather than directly to users). Figure 5.2 illustrates the situation where multiple end applications interact with multiple services; the use of layers maintains structure.

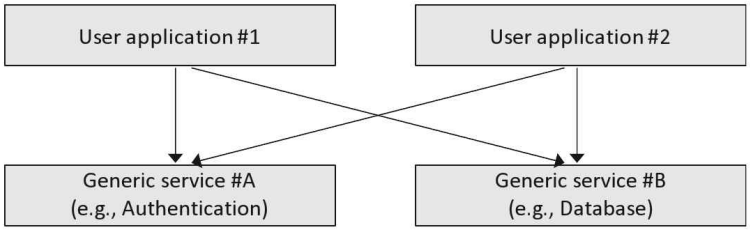


FIGURE 5.2

Layers facilitate logical separation of types of components and provide structure for interaction between components of systems.

- In addition to distributed applications and services, there is the system software that includes operating systems and various specialized components such as device drivers. The system software itself can be very complex in terms of the number of components it comprises and the range of functionalities it provides.
- The layers provide a structure, in which components occupy the same layer as similar components; sit in higher layers than the components they control or coordinate or use the services of; and sit below components that coordinate them or that they provide service to.
- There is a natural coupling between adjacent layers. There is also a decoupling of nonadjacent layers that contributes to flexibility and adaptability; this encourages clear division of roles and functionality between categories of components and also encourages the use of standardized and well-documented interfaces between components (especially at the boundaries between layers).

5.2.5 HIERARCHICAL ARCHITECTURES

A hierarchical architecture comprises multiple levels of components connected in such a way as to reflect the control and communication relationships between them. The higher node in a particular relationship is referred to as the parent, while the lower node in the relationship is referred to as the child. A node that has no nodes below it is termed a leaf node. Hierarchal organization has the potential to achieve scalability, control, configuration, and structure simultaneously.

The hierarchy illustrates the relative significance of the components, those being higher in the structure typically having more central or key roles. This maps onto the way most businesses are organized, with senior central managers higher in the structure, then department leaders, while workers with very specific well-defined areas of responsibility (functionality) occupying the leaf nodes. In hierarchically organized businesses, the normal communication is between workers and their manager (who is at the next level up in the structure). If something has to be passed up to a higher level, it is normally relayed via the managers at each level and not passed directly between nonadjacent layers. This has the effect of maintaining a clear control and communication regime that is appropriate for many large organizations that would otherwise struggle with scalability issues (e.g., imagine the complexity and confusion that could result if all workers in a large company contacted the senior management on a daily basis). Another example of the use of hierarchy is in the organization of large groups of people, such as armies, police forces, and governments. In such cases, it is important to

have central decision making for the highest-level strategies, while more localized decisions are made lower in the structure. The result is hopefully a balance between a uniform centrally managed system with respect to major policy while allowing local autonomy for issues that do not need the attention of the senior leadership.

However, rigid hierarchical structures can work against flexibility and dynamism; for example, the need to pass messages up through several layers causes overheads and delays. Reorganizations can be prohibitively complex especially where components are heavily interconnected.

Figure 5.3 illustrates common variations of hierarchical structures. Broad (or flat) trees are characterized by having few levels and a large number of components connected at the same level to a single parent node; this can affect scalability because the parent node must manage and communicate with each of the child nodes. Deep trees have many layers with relatively few nodes at each layer. This can be useful where such a detailed functional separation is justified, but in general, the additional layers are problematic because they add complexity. The main specific problems are increases in the communication costs and the latency of communication and control, as on average messages have to be passed up and down more levels of the tree. In general, a balanced tree (i.e., a balance is achieved between the breadth and the depth of the tree) is close to optimal in terms of the compromise between short average path lengths for communication and also the need to limit the number of subordinates at any point in the tree for manageability. A binary tree is a special case in which each node has (at most)

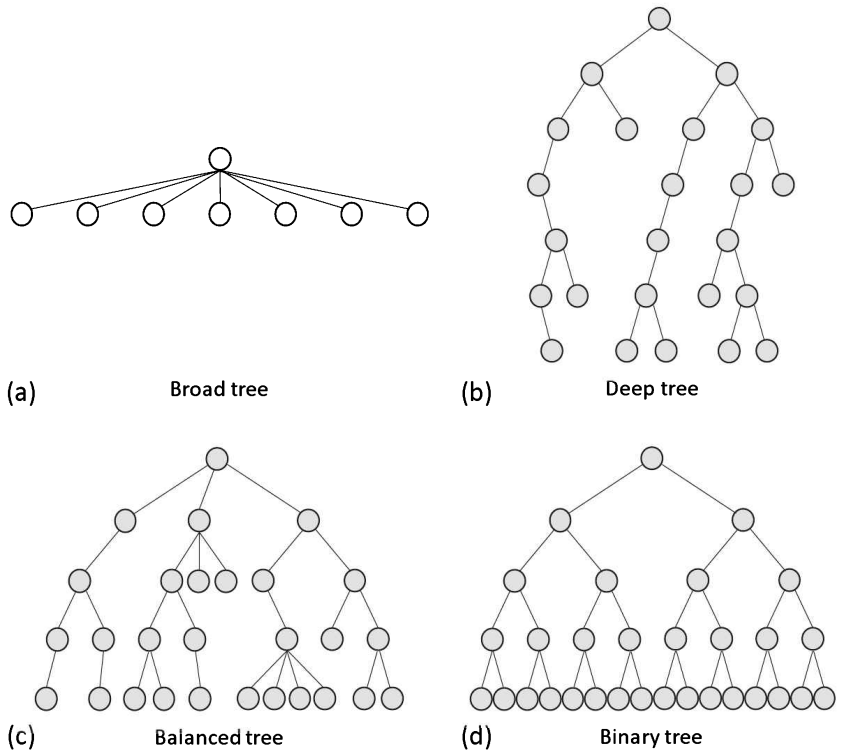


FIGURE 5.3

Hierarchical structures.

two subordinates. These are more likely to be found within data structures than in system component architectures but are included here for completeness.

5.3 HETEROGENEITY

Homogeneous systems are those in which all computers are the same, in terms of their hardware and configuration, resources, and operating system. These systems exist when, for example, a company or university equips an office or a laboratory with identical computers or a bespoke sensor system with identical sensor nodes is deployed. However, in the general case, processing systems are not identical and the various differences between them can impact on the configuration and management effort and can cause problems for interoperability, of varying complexity.

Heterogeneity is a very important concept for distributed systems, both in terms of a purposeful architectural feature to achieve a certain performance or behavior and in terms of it being one of the main challenges for interoperability and code portability.

5.3.1 DEFINITIONS AND SOURCES OF HETEROGENEITY

There are three main causes of heterogeneity, these being technological, intended, and accidental. Technological advances lead to new platforms or better resourced upgrades of earlier platforms. Other technological factors include advances in operating systems and programming languages as well as the occasional introduction of new network protocols. Heterogeneity is often intentionally introduced through design or configuration. For example, when more powerful platforms are used to host services, while users have less powerful workstations to access the services. A second example is where an operating system such as Microsoft Windows is chosen for the access workstation because of the popular user interface it provides, while Unix is chosen for the service-hosting platforms due to it having better configurability. A third example is where a service is hosted on a conventional static computer, while mobile computing devices, with completely different platforms and resource levels, are used to access the service. Heterogeneity is accidentally introduced, for example, when there are staged upgrades across hardware systems or when individual machines are enhanced or when different versions of the same base operating system are installed on different computers. Even an automated online update of the operating system that occurs on one computer but not on another potentially introduces heterogeneity if the behavior of one is changed in a way that the behavior of the other is not.

There are three main categories of heterogeneity: performance, platform, and operating system.

Performance heterogeneity arises from differences in resource provision leading to different performance of computing units. Common resource characteristics that lead to performance heterogeneity include memory size, memory access speed, disk size, disk access speed, processor speed, and network bandwidth at the computer-to-access network interface. In general, it is unlikely that any two computers will have identical resource configuration, and thus, there will usually be some element of different performances. There are many ways in which performance heterogeneity arises through normal system acquisition, upgrade, and configuration of hosted services such as file systems. Even buying computers in two batches a few months apart can lead to differences in the actual processor speed, memory size, or disk size supplied.

Platform heterogeneity (also termed architecture heterogeneity) arises from differences in the underlying platform, hardware facilities, instruction set, storage and transmission byte ordering, the number of actual processors within each machine's core, etc.

Operating system heterogeneity arises from differences that occur between different operating systems or different versions of operating systems. These include differences in the process interface, different types of thread provision, different levels of security, different services offered, and the extent to which interfaces are standard or open (published) versus proprietary designs. These differences affect the compatibility of, and challenges involved with porting of, software between the systems.

5.3.2 PERFORMANCE HETEROGENEITY

Figure 5.4 shows three computers, all having the same hardware platform and operating system. Computers A and B have the same level of resource and are thus performance homogeneous. Computer C has different levels of resource, having a slower CPU processing speed, more primary memory, and a smaller hard disk. All three computers will run the same applications, with the same operating system interface and support and the same executable files (as the hardware instruction set is the same in all cases). However, applications will perform differently on computer C than on A or B. Applications requiring a lot of file storage space are more likely to exceed capacity on computer C, and compute-intense applications will take longer to run. However, applications requiring a lot of primary memory may perform better on computer C. This is a simplified example that only considers the main resource types.

The example illustrated in Figure 5.4 represents the very common scenario that arises from piecemeal upgrade and replacement of systems, resulting in situations where you have a pair of computers configured with the same CPU and operating system, but, for example, one has had a memory expansion or been fitted with a larger or faster access time hard disk.

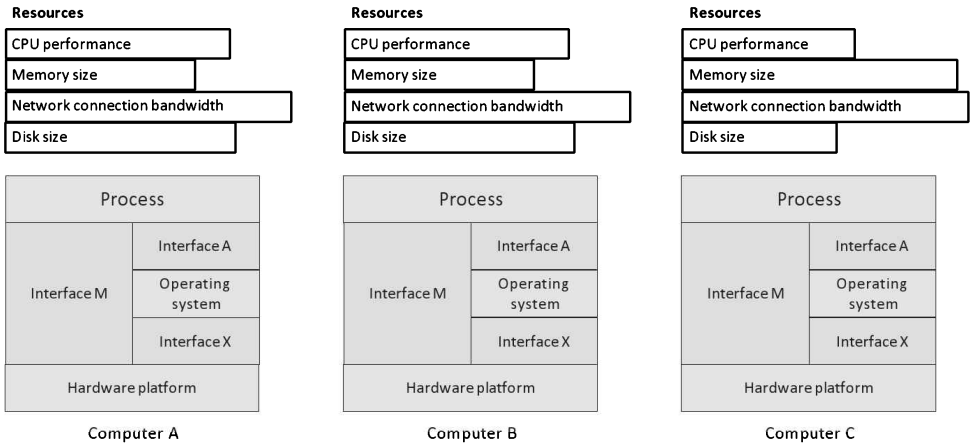


FIGURE 5.4

Performance heterogeneity.

5.3.3 PLATFORM HETEROGENEITY

Figure 5.5 illustrates a platform heterogeneous system. Each of the three computers D, E, and F has a different hardware platform, but all three run compatible variants of the same operating system.

Different platforms imply that the computers have different types of processor architecture, although different versions of the same processor family also represent a form of platform heterogeneity in cases where the run-time interface is different and thus the application code must be recompiled to run (e.g., one version of the processor family may support additional hardware instructions not available in the others). To some extent, performance heterogeneity (as a side effect) is inevitable in such cases because the CPUs may operate at different processing speeds (the number of instructions executed per second) or have different levels of internal optimization (such as branch prediction techniques to allow additional machine code instructions to be prefetched into cache ahead of execution). The interface between the platform and the operating system is different in each case; note the different interfaces X, Y, and Z, which means that different versions of the operating system are needed for each platform. If the operating system is the same, as in the three scenarios shown, the process interface to the operating system will remain the same; that is, it will have the same system calls to interface with devices, files, programmable timers, threads, the network, etc. Linux provides a very good example of this scenario in which the same operating system runs on many different platforms but provides the same process interface in each case. However, even though the source code for the applications could be the same in the three scenarios shown in Figure 5.5, the code will have to be compiled specifically for each platform, as the process-to-machine interface is different in each case (see interfaces M, N, and P). The differences, at this "machine code" level, may include a different instruction set, a different register set, and a different memory map.

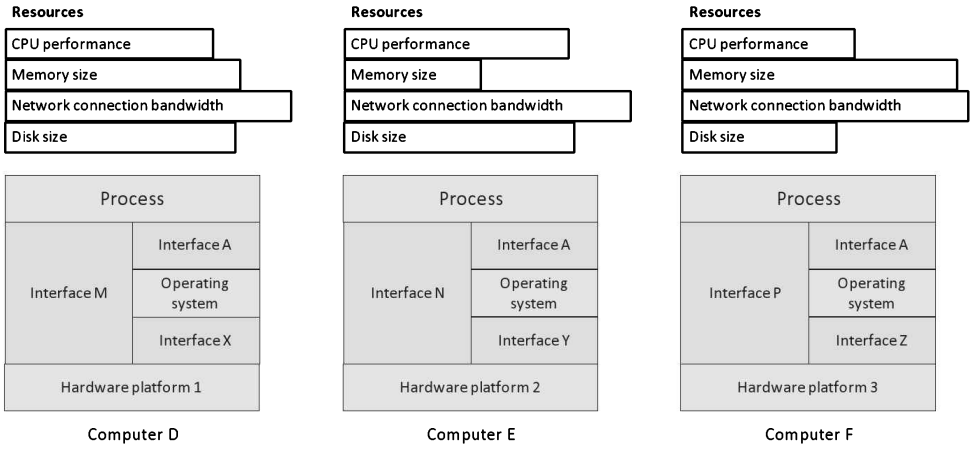


FIGURE 5.5

Platform heterogeneity.

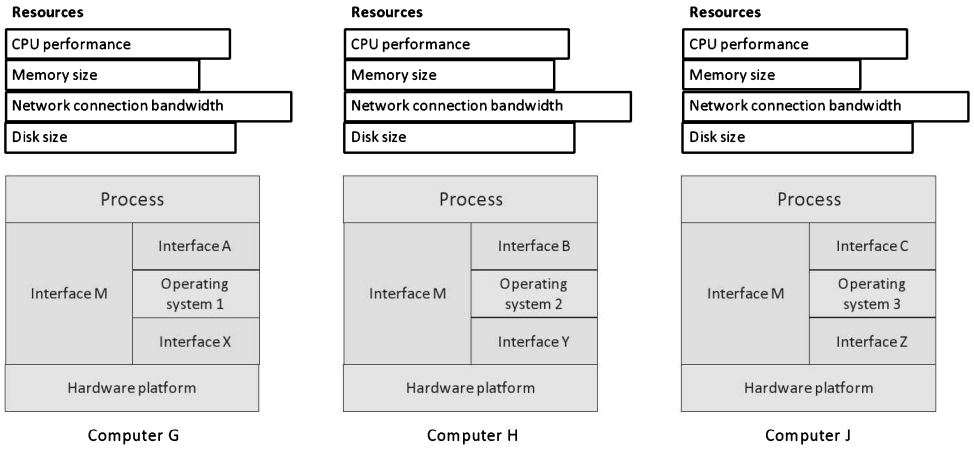


FIGURE 5.6

Operating system heterogeneity.

5.3.4 OPERATING SYSTEM HETEROGENEITY

Figure 5.6 illustrates operating system heterogeneity. In the scenario shown, all three computers G, H, and J have different operating systems but the same hardware platform. The three different operating systems will each have to use the facilities of the same platform type, so there will be differences in the interface between the operating system and the hardware, that is, interfaces X, Y, and Z, although the differences are in the ways the different operating systems use the resources of the hardware, rather than the interface provided by the hardware per se (which is actually constant in the three configurations shown). Although the application process has the same business-level behavior in all three systems, the way this is achieved at the code level will differ slightly as the different operating systems will have different interfaces (fundamentally the set of system calls and the syntax and semantics of their use). For example, there may be different versions of the file handling commands available, with differences in the parameters passed to them. This is reflected in the figure by the different process-to-operating system interfaces A, B, and C. Transferring applications from one of these computers to another is called porting and would require modification of those parts of the code that are sensitive to the differences in the operating system calls supported. As the hardware platforms are the same in each case, the same machine code interface is provided in each case. Changing the operating system may affect the effective resource provision of the computer and so impacts on performance. The most notable way in which this tends to occur is in terms of the amount of memory taken up by the operating system and thus the amount of memory remaining for user processes to use. This is reflected in the figure by the different amount of memory resource shown for each computer (i.e., it is based on the effective resource availability after the operating system has taken its share of the resource).

5.3.5 IMPACT OF HETEROGENEITY

All forms of heterogeneity potentially impact on interoperability. Applications that operate in platform heterogeneous or operating systems heterogeneous systems thus rely on standardized communications between the platforms. This requires standard interfaces and protocols to ensure that the contents of

messages and the semantics of communication itself are preserved when a message is passed between two dissimilar computer systems. The sockets API that operates at the transport layer and has been discussed in detail in the communication view chapter provides a good example of a standard interface at the process level. The sockets API is supported by almost all operating systems and programming languages and across almost all platforms.

The TCP and UDP protocols (of the TCP/IP protocol suite) are very good examples of standard protocols that facilitate interoperability between almost any combinations of platforms, using the sockets interface to a process as the end point for communication. These two protocols are extremely important for the Internet; they are not only used directly to provide bespoke process-to-process communication (as, e.g., in the case study game) but also used to construct most of the higher-layer communication mechanisms such as RPC, RMI, middleware, and web services.

These protocols are examples of the few invariants in distributed systems that have stood the test of time. The extent that they are embedded into such a wide range of essential services and higher-layer communication mechanisms reinforces their value as standards. It is relatively safe to assume that support for these protocols will remain for many years to come: future-proofing communications based on these protocols.

Performance heterogeneity is very common, to the extent that it is sometimes difficult to find computers with identical performance. However, if this is the only type of heterogeneity present, then applications will generally operate correctly on any of the host computers; but the overall speed and responsiveness will vary depending on the actual resource provision. Ideally, the functional split across software components and the subsequent deployment of these components onto processing nodes should match resource provision to ensure that the intended performance is achieved (but this is subject to the limitations of the design-time knowledge of the eventual run-time systems).

Platform heterogeneity is increasingly common, especially with the recent explosion of popularity of mobile devices including smart phones and tablets. Users demand applications that operate the same on their personal computer (PC), their phone, and their tablet, fundamentally because there is a desire to be "always connected" to favorite applications whether at home, in the office, or traveling between. It is not always possible to make applications identical on the different platforms, for example, the user interface on a smartphone with a touch screen cannot be identical to the user interface on a PC using a keyboard and mouse. The different platforms have different levels of resource, and this is sometimes evident in the way the software responds, for example, a fast-action game running on a smartphone with a small screen cannot in general be as impressive and responsive as the same game running on a PC that has been optimized for gaming with a very fast processor, expanded memory, and graphics accelerator processor.

Support for platform heterogeneity can add significant cost to software projects. Firstly, there will need to be a design approach that separates out the core functionality, which is device-independent, from the device or platform-specific functionality (typically mostly related to the user interface), which must be developed separately for each platform (see Section 5.13). The more platforms that are supported and the greater diversity between these, the greater the additional costs will be. Secondly, there are the additional testing costs. Each time the software is upgraded, it must be tested on all supported platforms; this in itself can be problematic in terms of the man power needed and the availability of a test facility in which the various platforms are all available. Some difficult to track down faults may occur on one platform only, requiring specific fixes that must then be tested to ensure they don't destabilize the product when on the other platforms. In some software projects, the testing team may be larger than the software development team.

Porting of code from one platform to another is less costly than ongoing maintenance and support for code across multiple platforms simultaneously, although it can still be challenging and potentially very expensive. In the simplest case where the operating system on each target platform is the same and the platforms themselves are similar, porting may only require that the source code (unchanged) must be recompiled to run on the new platform. However, if the operating system is also different, or where the platforms have significant differences, porting can require partial redesign.

Operating system heterogeneity introduces two main types of differences that affect processes: the first type being at the process-to-operating system interface and the second type being differences in the internal design and behavior of the operating systems. For the former, redesign of the sections of application code that make system calls (such as exec, read, write, and exit) and subsequent recompilation of the code may be sufficient. If the implementation of system calls is similar in the two operating systems, there may be no noticeable change in the application's behavior. However, for the latter, there potential problems in that the two operating systems may have different levels of robustness or one may have security weaknesses that the other may not have. For example, one operating system may be vulnerable to certain viruses and other threats that the other is immune to. There could also be effects on performance of applications due to differences in operating system behaviors including scheduling and resource allocation.

5.3.6 PORTING OF SOFTWARE

Porting applications from one system to another can be very complex because the two host systems can be very different in the various ways discussed above. The end result may be applications that are functionally similar, but some aspects such as the user interface or the response time may be noticeably different due to resource availability. Browsers provide a good example where essentially, the same functionality is available but with different look and feel on the various different platforms they run on. Browser technology was initially established on the general-purpose PCs (laptop and desktop computers) for many years but has now been adapted to operate in the same well-understood ways on mobile devices, which have different processors and operating systems and generally fewer resources in terms of memory, processing speed, smaller displays, and often lower bandwidth network connections than their PC counterparts.

A virtual machine (VM) is a software program that sits between application processes and the underlying computer. As far as the computer's scheduler is concerned, the VM is the running process. The VM actually provides a run-time environment for applications that emulates the real underlying computer; we can say that the applications run in, or on, the VM. Because the application processes interact with the VM instead of the physical computer, the VM can mask the true nature of the physical computer and can enable programs compiled on different platforms to run. By providing a mock-up of the environment the application needs to run on, the VM approach avoids the need for porting per se. The VM approach is key to the way in which Java programs are executed. An application is compiled to a special format called Java bytecode; this is an intermediate format that is computer platform-neutral. A Java-specific VM (JVM) interprets the bytecode (i.e., it runs the instructions from the bytecode) the same regardless of the physical environment; therefore, portability is much less of an issue for Java programs generally than it is for programs written in other languages. The VM (or JVM) approach does of course require that a VM (or JVM) is available for each of the platforms that you wish to run the applications on. The VMs (or JVMs) themselves are platform-specific; they are compiled and run as a

regular, native, process on whichever platform they are designed for. VMs and the JVM are discussed in more detail later in this chapter.

Middleware provides various services to applications that decouple some aspects of their operation from the underlying physical machine, especially with respect to access to resources that may be either local (on the same computer) or remote. In this way, middleware enables processes to execute in the same logical way regardless of their physical location. For example, the middleware may automatically locate resources the process needs or may automatically pass messages from one process to another without them having to have a direct connection or to even know the location of each other (refer to Section 5.5.1 in this chapter). The middleware may be implemented across different hardware platforms so that a process running on one platform type may transparently use resources located at a computer with a different platform type. Middleware does not however actually execute applications' instructions in the way that a VM does, and thus, it does not solve the portability problem directly. However, because middleware can enable the remote use of resources that are on different platforms, as well as communication between processes on different types of platforms, it offers an indirect solution to portability, that of transparent remote access, without the process actually moving to the other platform. An overview of middleware is provided later in this chapter and a more detailed discussion is provided in Chapter 6.

5.4 HARDWARE AND SYSTEM-LEVEL ARCHITECTURES

Distributed applications comprise software components that are dispersed across the various computers in the system. In order for these components to operate as a coherent single application, as opposed to isolated components doing their own thing, there need to be some means for the components to communicate. For this to be possible, there must be some form of connection between the underlying processors, on which the communication support software runs.

There are two fundamentally different approaches to connecting the processors together. Tightly coupled architectures are those in which the processors are part of a single physical system, and thus, the communication can be implemented by direct dedicated connections. In such systems, the processors may be equipped with special communication interfaces (or channels) designed for direct interconnection to other processors, without the need for a computer network. The processors share the resources of the computer, including the clock, memory, and IO devices.

Stand-alone computer architectures are those in which each processor is the core of a complete computer with its own dedicated set of resources. The PC, smartphone, tablet, and laptop computers are all examples of this class of computer. Stand-alone devices need an external network to communicate. There needs to be a network interface connecting each computer to the network as well as special communication software on each computer to send and receive messages over the network. This form of connecting the computers together yields a less tight and more flexible coupling; hence, it is termed loose coupling.

5.4.1 TIGHTLY COUPLED (HARDWARE) SYSTEMS

The main characteristic of tightly coupled systems is that they comprise a number of processor units integrated together in the same physical computer. This means that several threads of program code can run at the same time, that is, in parallel, since each processor can execute one instruction in each

timestep. In these architectures, the communication channels between processor units are very short and can be implemented using similar technology to that of the processor units, meaning that communication can take place at similar speeds to memory accesses. In fact, since the processors usually have shared access to at least part of the system memory, it is possible for the program threads to actually communicate using the memory. For example, if one process writes a new value to a particular variable stored in a shared memory location, all of the other processes can read the variable, without the need to specifically send a message to each process. The main advantages of this form of communication are that it has the same near-perfect reliability as memory accesses and that it does not become a bottleneck in terms of performance; writing to memory is effectively the same operation as sending a data value to another processor. This means that parallel applications can be developed to solve algorithms in which there is high communication intensity (a high rate of communication between the processors). In contrast, such applications do not perform so well on loosely coupled architectures due to the presence of an external network technology that operates at lower speeds than memory accesses, has higher latency due to greater physical distances, and is also less reliable. In tightly coupled architectures, there is usually a single shared clock, and thus, it is possible to synchronize processes accurately in terms of the application-level events and resulting actions carried out. Each processor executes instructions at exactly the same rate; there can be no relative clock drift when there is only a single clock.

5.4.2 LOOSELY COUPLED (HARDWARE) SYSTEMS

This book focuses on distributed applications that run on loosely coupled systems. These systems consist of a number of self-contained computers able to function independently of any others. A perfect example of this is the PC I'm using to write the book. The computer has its own power supply, processor, clock, memory, hard disk storage, operating system, and IO devices. However, the fact that the computer is self-contained does not necessarily mean that the computer can do what I require of it in isolation. Most of the applications that are used in modern business, as well as in hobbies and entertainment and social media, require access to data held at other computers and also require a means to communicate with other users, via the computers that they are using. Therefore, almost every computer has a network connection to support the communication requirements of distributed applications and data. This form of coupling is termed “loose” because the network is external to the computer itself. The communication over networks is slower and less reliable than in tightly coupled systems. However, the communication in loosely coupled systems can be flexibly reconfigured so that a particular computer can be logically connected to any other that is reachable in the network.

Each computer has its own set of resources, which is beneficial in general, because the local scheduler has control of the way the resources (such as memory and the processor's computing cycles) are shared across the local processes. However, the fact that each computer also has its own clock that governs the precise rate at which instructions are executed and is also used to keep track of wall clock time (the actual real-world notion of time) introduces some challenges of synchronization when running distributed applications. For example, it is difficult to determine the global sequence with which a particular set of detected events occur (such as stock-trading transactions or deposits and withdrawals on a particular bank account) or to ensure the correct sequence of a set of critical actions is maintained (such as opening and closing control valves in an automated chemical production factory) if the actual processes associated with the events and actions are executing on different computers with imperfectly

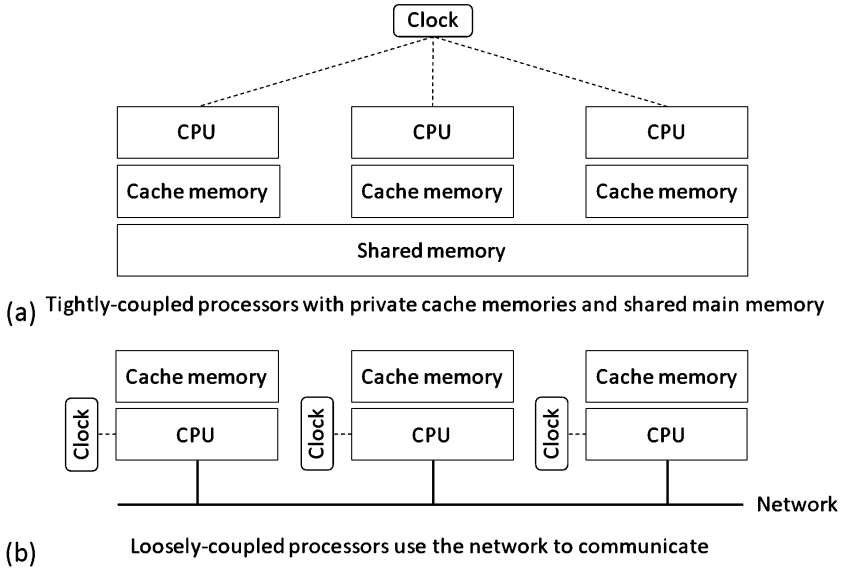


FIGURE 5.7

Tightly and loosely coupled hardware architectures.

synchronized clocks. The challenges of clock synchronization and some techniques to overcome these challenges are discussed in Chapter 6.

A further challenge arising for the use of interconnected but self-contained computers is that they can fail independently. Failures that occur when applications are quiescent do not corrupt data and thus are relatively easy to deal with. However, consider the failure of a computer during a data transfer. Whether or not the data become corrupted depends on a number of factors that include the exact timing with which the failure occurs and the extent to which the communication mechanism in use was designed to be robust with regard to maintaining data integrity.

Figure 5.7 illustrates the main concepts of tightly and loosely coupled hardware architectures. There are actually many variations of tightly coupled architectures—the main differences concerning access to memory and the way in which the processors communicate. In some designs, all of the memory is shared, while in others, there is also some private cache per processor (as shown), and some designs have dedicated communication channels between the processors.

5.4.3 PARALLEL PROCESSING

Parallel processing is a special class of distributed application, which is briefly described here for completeness.

A parallel application generally comprises a number of processes all doing the same calculations but with different data and on different processors such that the total amount of computing performed per unit time is significantly higher than if only a single processor is used.

The amount of communication that occurs between the processes is highly dependent on the nature of the application itself. In most parallel applications, the data at each processing node are not

processed in isolation; there is generally a need to interact with the computation of data values in other nodes, representing bordering regions of the application data space.

A classic example of a parallel processing application that most of us benefit from daily is weather forecasting, which uses specialized techniques such as computational fluid dynamics (CFD). The geographic area to be studied is divided up into small regions, which are further subdivided into smaller and smaller cells. The actual forecast of the weather at a particular time in the future, in a particular cell, is based not only on the history of weather conditions in the cell at times leading up to the forecast time but also on the weather conditions in the neighboring cells at each of those times, which influences the weather in our cell of interest. The weather conditions in the direct neighbor cells of our cell of interest are also affected by the conditions in the neighbors of neighbors cells at each timestep, and so it goes on. CFD works in iterations such that the state of each of the cells at a particular time point t_1 is computed based on the state of the cell at time t_0 and also the state of the neighboring cells at time t_0. Once the new state of the cells at time t_1 is computed, it becomes the starting point for the next iteration to compute the state of the cells at time t_2. The total amount of computation is determined by the complexity of the actual algorithm and the number of cells in the model and the number of timesteps (e.g., the actual weather forecast for a region the size of the United Kingdom may work with a geographic data cell size of perhaps 1.5 km^2). The amount of communication depends on the extent of the dependency between the cells during the iteration steps of the algorithm.

A parallel application that has a high ratio of computation to communication (i.e., it does a lot of computation in between each communication episode) may be suited to operation on loosely coupled systems. However, an application where the communication occurs at a high rate with a low amount of computation in between the communication is only suitable for execution on specialized tightly coupled architectures. If executed on a loosely coupled system, such applications tend to progress slower, possibly even slower than an equivalent nonparallel version on a single processor. This is because of the communication latency and that the total communication requirement can exceed the communication bandwidth available, so more time is spent waiting for the communication medium to become free and for messages to be transmitted than actually performing useful processing.

5.5 SOFTWARE ARCHITECTURES

In distributed applications, the business logic is split across two or more components. A good design will ensure that this is done to achieve a "separation of concerns" to the greatest extent possible. By this, it is meant that it is ideal to split the logical boundary of components on a functional basis rather than on a more abstract basis. If the logic is divided into components on an arbitrary basis (e.g., perhaps to try to keep all components the same size), then there will likely be more communication and interdependence between the resulting components. This can lead to a more complex and fragile structure because problems affecting one component also affect directly coupled components and can be propagated through a chain of components.

A design in which component boundaries are aligned with the natural functional behavior boundaries can result in a much less coupled structure in which individual functionalities can be replaced or upgraded without destabilizing the entire architecture, and whereby faults can be contained, such that a failed component does not lead to a domino-effect collapse. Dividing the business logic along functional lines also makes it possible to target robustness and recovery mechanisms (such as replication of

services) to specific areas of the system that either are more critical to system operation or perhaps are more sensitive to external events and thus more likely to fail.

The way in which the business logic is functionally divided across the components is in many cases the single most significant factor that influences the performance, robustness, and efficiency of distributed applications. This is because it affects the way in which binding between components takes place and the extent of, and complexity of, communication. If done badly, the intensity of communication may be several times higher than the optimal level.

Some functionality however needs to be implemented across several components, due to the way the application is structured or operates. For example, the business logic of a client-server (CS) application may distribute the management and storage of state information across the two types of component. State that is needed only on the client side may be managed within the client component (improving scalability because the server's workload per client is reduced), while shared state needs to be managed and stored at the server side because it is accessed by transactions involving several clients.

5.5.1 COUPLING BETWEEN SOFTWARE COMPONENTS

There are a variety of ways in which multiple software components (running as processes) can be connected together in order to achieve a certain configuration or structure. The application-level business logic and behavior are thus defined by the collective logic of the components and communication between them. The term "coupling" is used to describe the ways in which the various processes are connected together in order to achieve the higher business-level (logical) connectivity. The nature of this coupling is a critical aspect of successful design of distributed systems.

As discussed above, excessive connections and direct dependencies between components are generally problematic in terms of scalability and robustness. Direct dependencies also inhibit dynamic reconfiguration, which is increasingly important in highly complex feature-rich applications and in application domains in which the operating environment is itself highly dynamic.

There are several forms of coupling as explained below. Whether the coupling is tight or loose is determined by the extent of run-time flexibility built in at design time.

Tight (or fixed) coupling is characterized by design time-decided connections between specific components. Explicit references to other components introduce direct dependencies between the components, which means that the application can only function if all of the required components (as per its fixed design-time architecture) are available. If one component fails, the other components that depend on it either fail or at least cannot provide the functionalities that the failed component contributes to. Thus, tightly coupled design tends to increase sensitivity to failure, reduce flexibility, reduce scalability, increase the difficultly of maintenance, and inhibit run-time reconfigurability.

Loosely coupled (decoupled) components do not have connections with specific other components decided at design time. This form of coupling is characterized by intermediary services, which provide communication between components (such as middleware), and/or by the use of dynamic component discovery mechanisms (such as service advertisement). The components are coupled with indirect references. This is a run-time flexible approach as it is possible for the intermediary service to modify the references to other components based on the at-the-time availability of other components; thus, components are not directly dependent on specific instances of other components or processes. For example, a client could be mapped to one of many instances of a service, depending on availability at the time the service request is made, thus making the application robust with respect to the failure of

individual service components. Intercomponent mapping can also be based on a description of required functionality rather than based on a specific component ID, thus, at run time, the intermediary (such as middleware) can connect components based on a matching of what one needs and what the other provides.

As loose coupling uses external services or mechanisms to provide the mapping between pairs of components, it has the potential to make applications access and location transparent. Loosely coupled applications are easier to maintain as upgrades can be supported by changing the run-time mapping to swap the new version of a component into the position held by the old version. Loosely coupled applications are also extensible as new components with new functionality can be added without redesigning the other components or recompiling the system. See Figure 5.8.

Figure 5.8 illustrates the advantages of loose coupling. The dynamic mapping facilitates location transparency as the client does not need to know the location of the service component. This enables automatic remapping to a different service instance if the currently used one fails (as between times t_1 and t_3 when server instance A of the service fails and the client requests are remapped to instance B of the same service) and also remapping to an upgraded version of a service (as between times t_3 and t_5 when the service is upgraded and subsequent client requests are mapped to a new version 2 server instance). Access transparency is also provided in some cases where the connectivity service handles differences in the application service interfaces (arising, e.g., during a service upgrade) so that the client components remain unchanged. This is very important where there are high numbers of clients deployed and upgrading them all in step with each server upgrade would be expensive in terms of logistics, time, and effort and risks the situation where different versions of the client could be in use in the system at the same time.

Logical connections can be direct between communicating components or can be facilitated indirectly by intermediate components:

Direct coupling is characterized by the fact that the process-to-process-level connections correspond with the application's business-level communication. The transport layer logical connections map directly onto the higher-level connectivity. For example, there may be a direct TCP or UDP connection between the business-level components.

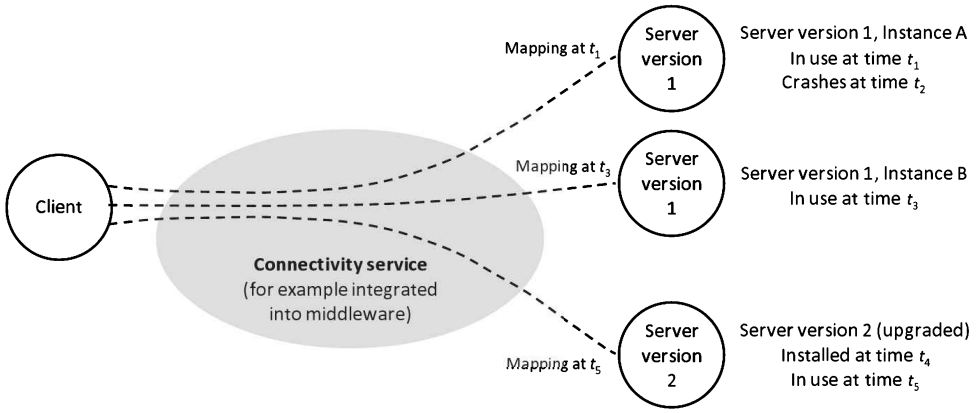


FIGURE 5.8

Dynamic mapping in loosely coupled systems.

Indirect coupling describes the situation where components interact through an intermediary. A stock-trading system provides an example where clients have private connections but see the effects of other clients' trades (in the form of changes in the stock price), which may lead to further transactions. Another example is provided by a multiplayer game hosted on a central server, where the game clients are each aware of the other's presence at the application level (they are logically connected by the fact that they are opponents in the same game) but are not directly connected together as components. Each client is connected only to the server and any communication between the clients is passed to the server and forwarded to the other client. The use-case game application provides a useful example of this form of coupling. A further example is e-mail, in which people use their e-mail clients to send e-mail messages to each other (the logical connection is in terms of the e-mail conversation). The e-mail clients each connect to the users' respective e-mail server, which holds their mail inbox and also sends outgoing mail; see Figure 5.9.

Figure 5.9 uses e-mail to illustrate indirect coupling. The users (actually the e-mail client processes they use) have a logical connection in the passing of e-mail messages at the application level. The e-mail clients are not connected directly, not even if both users have the same e-mail server. In the figure, there are two intermediaries between the e-mail clients. Each client is directly coupled to its respective e-mail server, and the two servers are directly coupled to each other. Notice that this does not affect whether the directly coupled components are tightly or loosely coupled; this depends on how the association between the components is made (e.g., there could be an intermediary service such as middleware that provides dynamic connectivity).

Isolated coupling describes the situation where components are not coupled together and do not communicate with each other although they are part of the same system. For example, a pair of clients each connected to the same server do not need to communicate directly or even be aware that the other exists. Consider, for example, two users of an online banking system. Their client processes each access the banking server, but there is no logical association between the two clients; they each have independent logical connections with the bank. The e-mail example shown in Figure 5.9 provides the framework for another example: consider two users who do not know each

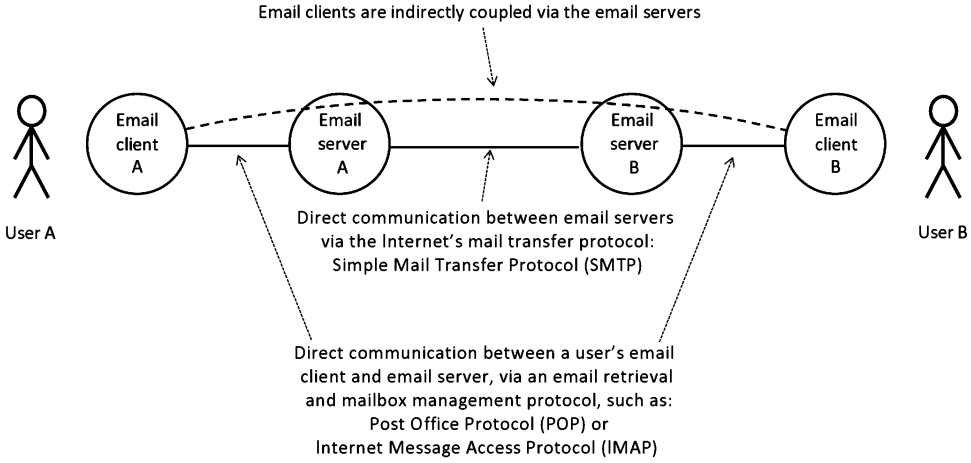


FIGURE 5.9

Sending e-mail involves several components. E-mail clients are indirectly coupled.

other and never send e-mails to each other. The respective e-mail client processes are each part of the same system but have no logical association in this case, and thus, the processes are not coupled together.

Figure 5.10 illustrates the coupling variations possible, based on a CS application as an example. Part (a) shows direct coupling in which the business-level connectivity is reflected directly by the process-to-process connection. This is a very common configuration in small-scale applications. Part (b) shows indirect coupling using a specific component as an intermediary that forwards communication between the connected processes. The central component is part of the application and participates in, and possibly manages, the business-level connectivity; this is reflected in the figure by the business logic connection being shown to pass through the server component. The use-case game provides an example of this, as the server needs to observe game-state changes, update its own representation of the game state, and forward the moves made by one client to the other. The server also has to check for game-end conditions, that is, one client has won or it was a draw and also needs to regulate the turn-based activity of the clients. Part (c) comprises parts (a) and (b) to illustrate

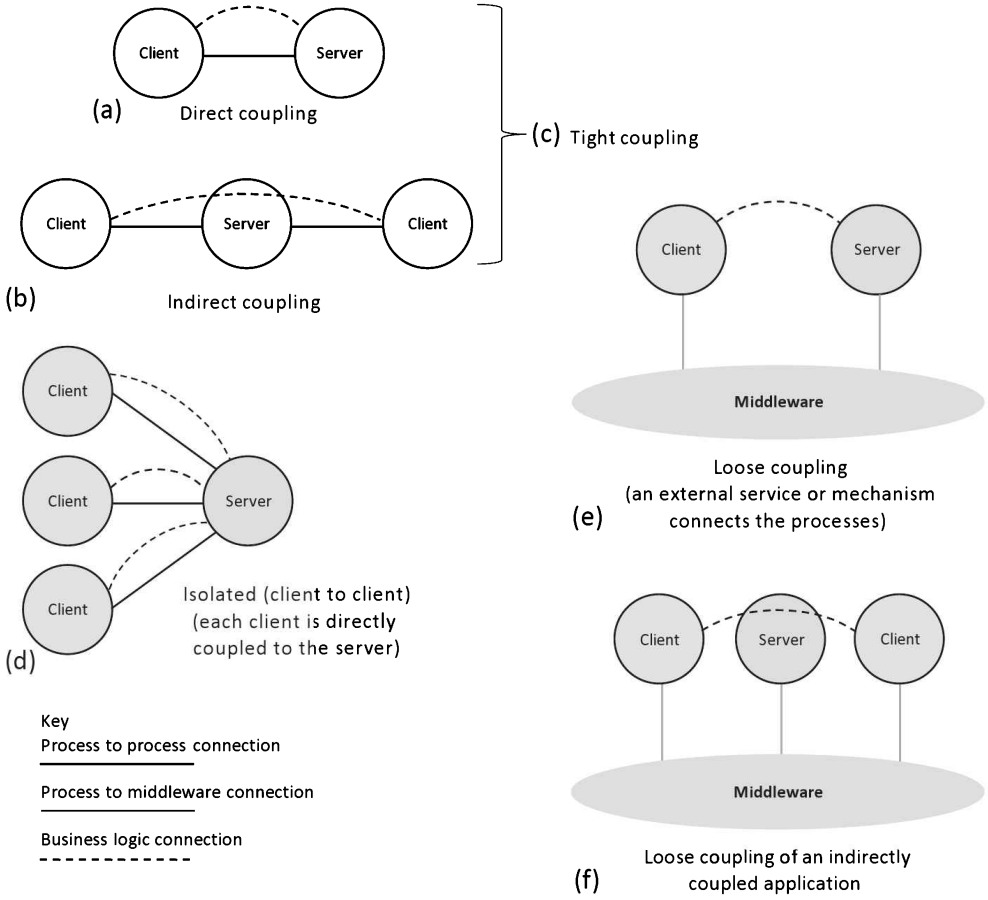


FIGURE 5.10

Coupling variations, illustrated in the context of a client-server application.

tight coupling between components, that is, where the components connect directly to specific other components with which they have a business logic relationship. The application-level architecture is "designed into" these components such that they connect to other components in a predecided way. Part (d) shows that clients using a common service, where each client interaction with the service is private to that specific client, are isolated (i.e., the clients are not coupled to each another) at the business level. Each client obtains service from the server without interaction with, or knowledge of the presence of, the other clients. Part (e) shows how an external connectivity mechanism or service, such as middleware, can be used to provide connectivity between processes without them having to actually form or manage the connectivity directly between themselves. The connectivity mechanism is not part of the application and does not participate in the business-level connectivity; its role is to transparently pass messages between the two components without knowledge of their meaning or content. This is a much more flexible means of connecting processes and is very important for large-scale or dynamically reconfigurable applications. Each of the direct and indirect coupling modes can be achieved in loosely coupled ways, as confirmed by part (f) of the figure, where the messages are passed through the middleware instead of directly between components, but the component relationships at the business logic level are unchanged.

Scalability is in general inversely related to the extent of coupling between components due to the communication and synchronization overheads that coupling introduces. Therefore, excessive coupling can be seen as a cost and should be avoided by design where possible. Thus, for large systems, scalable design will tend to require that most of the components are isolated with respect to each other and that each component is only coupled with the minimum necessary set of other components.

Figure 5.11 shows two different possible couplings between components in the same system. Part (a) shows a tree configuration that tends to be efficient so long as the natural communication channels are not impeded by having to forward messages through several components. This requires good design so that the case where a message is passed from a leaf node up to the root of the tree and down a different branch to another leaf is a rare occurrence and that most communication occurs between pairs of components that are adjacent in the tree. Part (b) shows a significantly more complex mapping, which introduces a higher degree of intercomponent dependency. It may be that the complexity of the connectivity is inherent in the application logic and cannot be further simplified, although such a mapping should be carefully scrutinized.

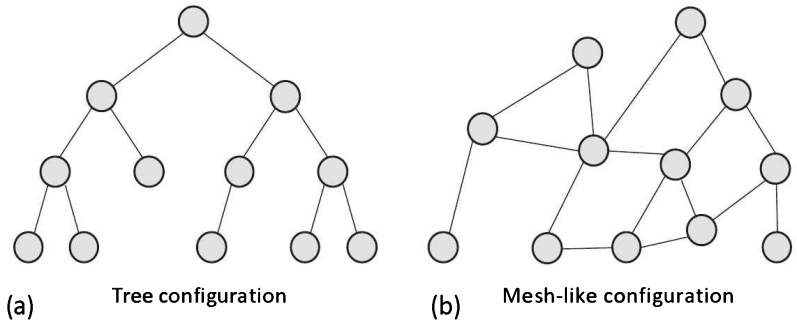


FIGURE 5.11

Different complexities of component coupling.

5.6 TAXONOMY OF SOFTWARE ARCHITECTURE CLASSES

The various activities performed by an application can be generically divided into three main strands of functionality: The first strand is related to the user interface, the second strand is the business logic of the application, and the third strand is functionality related to storage of and access to data. These are three areas of functionality that are usually present in all applications to some extent, and if their descriptions are kept general enough, they tend to cover all common activities. This broad categorization is very useful as a means of describing and comparing the distribution of functionalities over the various components of a system. Note that the description in terms of these strands is purposely kept at a high level and is thus more useful to describe and classify the approach taken in the distribution (i.e., in terms of the overall design and behavioral effect of the design) rather than to describe specific features of a design in any detail.

5.6.1 SINGLE-TIER APPLICATIONS

A single-tier application is one in which the three main strands of functionality are all combined within a single component, that is, there is no distribution. Such applications tend to be local utilities that have restricted functionality. In terms of business applications, single-tier design is becoming quite rare as it lacks the connectivity and data-sharing qualities necessary to achieve the more advanced behaviors needed in many applications.

Figure 5.12 illustrates the mapping of the three main strands of functionality onto a single application component. Such applications are sometimes referred to as stand-alone applications.

5.6.2 TWO-TIER APPLICATIONS

Two-tier applications split the main strands of functionality across two types of component. The most common example of a two-tier application is the CS architecture (discussed in detail later).

Figure 5.13 shows some of the possible variations of functionality distribution for CS applications. Note that the figure is illustrative only and is not intended to provide an accurate indication as to the proportion of processing effort dedicated to each of the functional strands.

Peer-to-peer applications (also discussed later in detail) can be considered a hybrid between the single-tier and two-tier approaches. This is because each peer instance is essentially self-contained and thus has elements of all of the three functionality strands. However, to cooperate as peers, there must

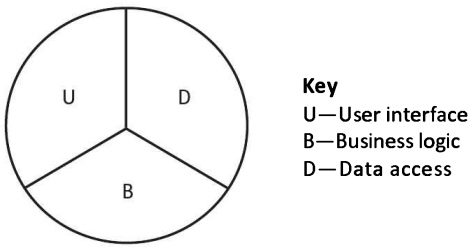


FIGURE 5.12

Single-tier design places all of the main strands of functionality in a single-component type.

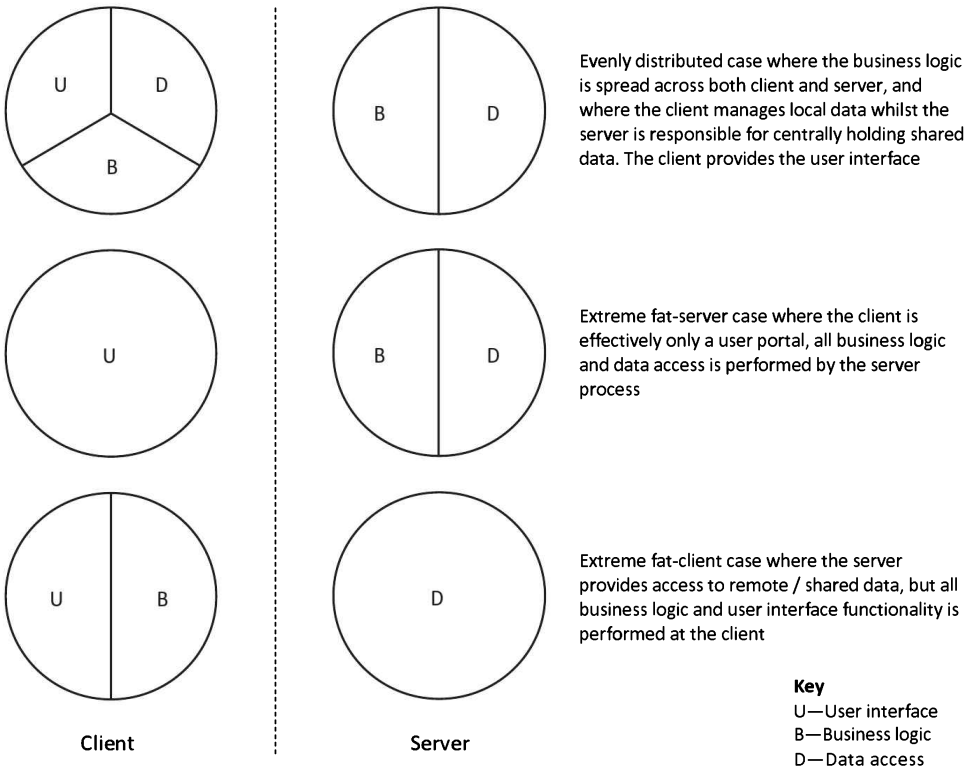


FIGURE 5.13

Two-tier design distributes the main strands of functionality across two-component types.

be some communication between instances, for example, it is common for peer-to-peer applications to facilitate data sharing, where each peer holds a subset of data and makes it available to other peers on demand (see Figure 5.14).

As Figure 5.14 implies, each peer contains elements of each functional strand and therefore can operate independently to some extent. For example, a music-sharing peer can provide its locally held files to the local user without connection to any other peers. Once peers are connected, they can each share the data held by the others.

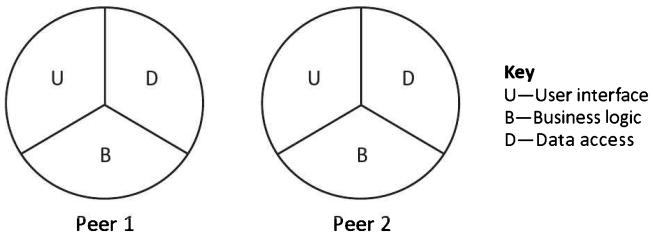


FIGURE 5.14

Peer-to-peer applications represented as a hybrid of single-tier and two-tier architecture.

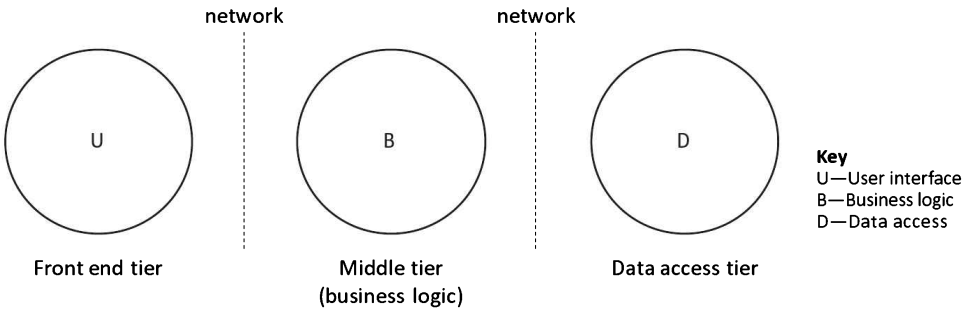


FIGURE 5.15

An idealized three-tier application.

5.6.3 THREE-TIER APPLICATIONS

Three-tier applications split the main strands of functionality across three component types. A general aim of this architecture is to separate each main strand of functionality into its own tier; see Figure 5.15. The potential advantages of splitting into three tiers include performance and scalability (because the workload can be spread across different platforms and replication can be introduced at the middle tier and/or the data access tier as necessary) as well as improved maintainability and extensibility (because, if the interfaces between tiers are well designed, functionality can be added at one tier without having to reengineer the other tiers).

Figure 5.15 illustrates an idealized three-tier application in which each of the three functional strands is implemented in a separate component to enhance scalability and performance. In reality, the separation of the functional strands is rarely this clean, and there will be some spread of the functional strands across the components. For example, there may be some data access logic and/or data storage in the middle tier, or there may be some aspect of business logic that is implemented in either the front-end or data access tiers because it may be more efficient or effective, depending on the actual application requirements.

5.6.4 MULTITIER APPLICATIONS

Many applications have extensive functionality and require the use of various additional services beyond performing the underlying business role. For example, a banking application may need functionality associated with security and authentication (of users as well as connected systems), interest rates, currency exchange rates, funds transfers between in-house accounts and to/from externally held accounts, calculation of fees and charges, and many others, in addition to the basic operations of managing funds within a bank account. Such functionally diverse systems cannot be built effectively using two-tier or three-tier approaches. In order to manage the complexity of the system itself and to ensure its extensibility, as well as to ensure the maintainability of the subcomponents, these systems need potentially very many component types, and the distribution of functional strands across the components is highly dependent on the needs of each specific business system.

Figure 5.16 illustrates the concept of multitier design. The same basic ideas of three-tier design apply, but the functionality is spread over more component types to give a modular and thus more scalable and extensible design. A main motivation for this approach is to manage (limit) the complexity of

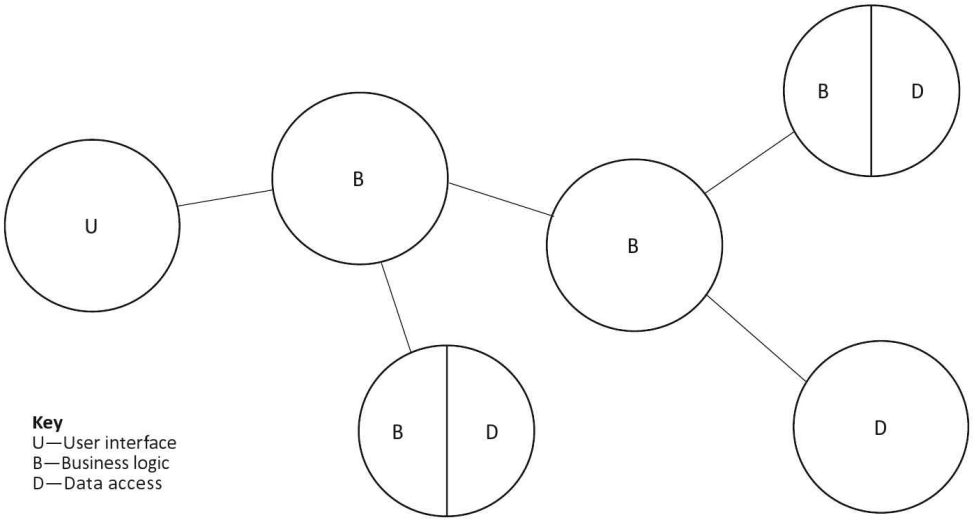


FIGURE 5.16

An example of functional distribution in a multitier application.

each component type and to achieve flexibility in the way the components are used and connected. For example, there may be some specific functions that are not required in all installations of the software; if the mapping of functionality onto components is done appropriately, then the relevant components can be omitted from some deployments. The modular approach also facilitates upgrade of individual components without disrupting others.

5.7 CLIENT-SERVER

CS is perhaps the most well-known model of distributed computing. This is a two-tier model in which the three main strands of functionality are divided across two types of component.

A running instantiation of a CS application comprises at least two components: at least one client and at least one server. The application's business logic is divided across these components that have defined roles and interact in a prechoreographed way to various extents in different applications. The interaction is defined by the application-level protocol, so for a very simple example, the client may request a connection, send a request (for service) message, receive a reply from the server, and then close the connection. Application-level protocols are discussed in Chapter 3.

5.7.1 LIFETIMES OF CLIENT AND SERVER

In most CS applications, the server runs continually and the client connects when necessary. This arrangement reflects the typical underlying business need: that the server should be always available because it is not possible to know when human users will need service and hence run the client. Essentially, the user expects an on-demand service. However, the user has no control over the server, and clients cannot cause the server to be started on demand. The client is usually shutdown once the user session is ended.

It is not desirable to leave the client components running continuously for several reasons that include the following:

1. It would use resources while active even when no requests are made, and there is no certainty that any further requests will ever be made.
2. Keeping clients running requires that their host computers are also kept running, even when they are not actually needed by their owners.
3. Many business-related applications involve clients handling user-private or company-secret information and generally require some form of user authentication (examples include banking, e-commerce, and e-mail applications); so even if the client component itself remains running, the user session has to be ended (and typically that also ends the connection with the server). A new user would have to be authenticated and a new connection established with the server, hence still incurring a large fraction of the total latency incurred when restarting the component from scratch.

5.7.2 ACTIVE AND PASSIVE SIDES OF THE CONNECTION

The component that initiates a connection is described as the active side, and the component that waits for connection requests is described as the passive side. (Think of actively striking up a conversation with someone, as opposed to being approached by someone else who starts up a conversation with you. You don't have to do anything initially except to be there; hence, your behavior is passive.)

Client on-demand connection to services is supported by two main features of the way servers usually operate: firstly, the fact that servers tend to be continually available (as discussed above) and, secondly, because they tend to be bound to well-known ports and can be addressed by fixed URLs (i.e., they can be located using a service such as DNS; see the detailed discussion in Chapter 6).

5.7.3 THE CS ARCHITECTURAL MODEL

CS is a logical architecture; it does not specify the relative physical locations of components, so they can be both on the same computer and on different computers. Each component runs as a process; therefore, the communication between them is at the transport layer (and above) and can be based on the use of sockets, the TCP or UDP protocols, and higher-level communication.

Figure 5.17 illustrates the general CS architecture, in a situation where two client processes are each connected to a single server. It is quite common that a server will allow many clients to be connected

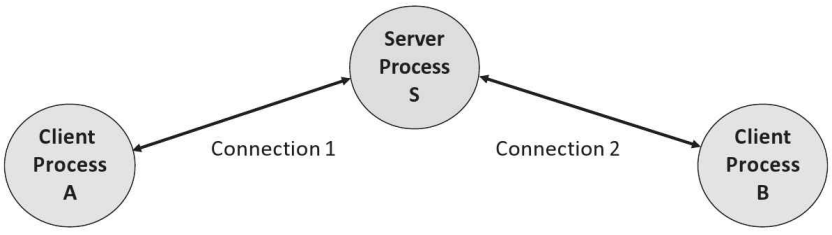


FIGURE 5.17

Client-server generalized representation.

at one time, depending on the nature of the application, but the communication is private between the server and each client.

CS is so named because normally, the server provides some sort of service to the clients. The communication is usually initiated by the client as the server cannot predict when a client will require service. Clients typically do not interact with each other directly as part of the CS application, and any communication that does occur between them is via the server. In many applications, the individual clients are unaware of each other's presence; in fact, this is a key design requirement for a wide variety of applications such as online banking, stock-trading, and media-steaming services.

In such applications, the business model is based on a private client-to-server relationship, which operates a request reply protocol in which the server responds to the requests of the client. The fact that there may be many clients simultaneously connected to the server should be completely hidden from the clients themselves; this is the requirement of concurrency transparency at the higher architecture level. However, a performance effect of other clients could become felt if the workload on the server is high enough that the clients queue for service on a timescale noticeable to the users (this is highly application-dependent). Techniques to ensure services remain responsive as load and scale increases include replication of servers, which is discussed in depth in this chapter and is also discussed from a transparency perspective in Chapter 6.

The clients in such applications can be said to be isolated from each other with respect to coupling, and they are independent of each other in terms of their business logic and function. The clients are each coupled with the server, either loosely or tightly depending on the design of the application-level protocol and the way that components are mapped to each other.

In contrast, a CS multiplayer game is a good example of an application where clients are logically connected via the game (their business logic) and thus are necessarily aware of each other and do interact but indirectly via the server (i.e., the clients are indirectly coupled to each other as discussed previously). In such an application, it is likely that the server will manage shared resources such as the game state and will control game flow, especially in turn-based games. In this case, the architecture fits the fat-server variant of CS; see the next section.

For interactive applications, the client is usually associated with human users (it acts as their agent). For example, an e-mail client connects to a remote e-mail service. The client provides an interface through which the user can request e-mail messages to be retrieved from the server or can send messages to the server to be sent as e-mails to other users.

Figure 5.18 illustrates a typical e-mail configuration and usage scenario. By holding the e-mail inbox centrally at a well-known location (which is identified by the domain part of the user's e-mail address URL), a user can access their e-mail from anywhere that they have a network connection. The user can access the server using one of several e-mail client programs running on different devices. The user interface and data presentation may be different in each case, but the actual data will be the same. Recipients of e-mail will not be able to determine which device, or e-mail client, was used to create the e-mail as all e-mails are actually sent out by the e-mail server.

Application servers are often hosted on dedicated computers that are configured specially for the purpose (e.g., it might have larger memory and faster CPU to ensure it can handle requests at a high speed and with low delay). In such cases, the computer itself is sometimes referred to as being the server, but in fact, it is the process that is running on the computer, which is actually the server.

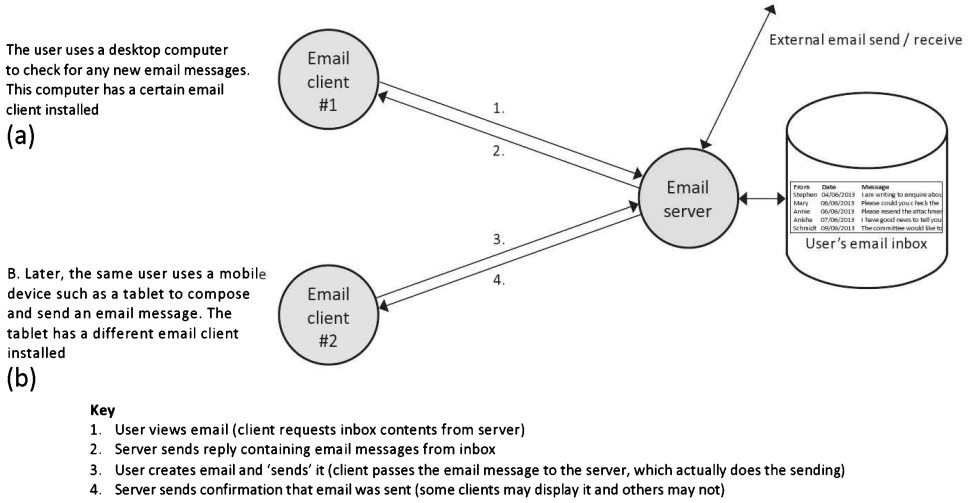


FIGURE 5.18

An e-mail application example of client-server configuration.

5.7.4 VARIANTS OF THE CS MODEL

As discussed earlier, there are three main strands of functionality in distributed applications, related with the user interface, the business logic, and access to and maintenance of data, respectively. Several architectural variants of CS arise, based on different distributions of these functional strands over the client and server components.

As the names suggest, fat server (also known as thin client) describes a configuration in which most of the work is performed at the server, while in fat client (also known as thin server), most of the work is performed in the client. Balanced variants are those where the work is more evenly shared between the components (these variations were illustrated in Figure 5.13). Even within these broad categories, there are various ways the actual strands of functionality are distributed, for example, the client may hold some local state information, the remainder being managed at the server. In such a case, this should be designed such that the clients hold only the subset of state that is specific to individual clients and the server holds shared state.

Fat server configurations have advantages in terms of shared data accessibility (all the data access logic and data are held at a central location), security (business logic is concentrated on the server side and is thus protected from unauthorized modification, and all data accesses can be filtered based on authorization), and upgradeability (updates to the business logic only have to be applied to the small number of server components, and are thus potentially much simpler than when large numbers of deployed clients have to be upgraded).

Fat client configurations are appropriate when much of the data are local to the individual users and do not require sharing with other clients. Fat client design is also ideal in situations where the business logic is somehow customized for individual users (this could be relevant, e.g., in some games and business applications such as stock trading). An important advantage of the fat client approach is scalability. The client does much of the work, such that the server is by implication "thin" (does less work per client) and thus can support more clients as the incremental cost of adding clients is relatively low.

CS applications that distribute the business logic and the data access functions across both client and server components can be described as balanced configurations. In such applications, there are two categories of data: that which is used only by individual clients, such as personal data, historical usage data, preferences, and configuration settings and that which is shared across all or many of the clients and thus needs to be held centrally. The business logic may also be divided across the two-component types such that processing that is local to the client and does not require access to the shared data is performed on the client's host (e.g., to allow behavior customization or to add specialized options related to the preferences or needs of the particular client or the related user). This also has the benefit of reducing the burden on the server's host, making the service more responsive and improving scalability. However, the core business logic that manipulates shared data and/or is expected to be subject to future change runs at the server side for efficiency and to facilitate maintenance and updates. The balanced configuration represents a compromise between the flexibility advantages of the fat client approach and the advantages of centralized data storage and management (for security and consistency) of the fat server.

5.7.5 STATEFUL VERSUS STATELESS SERVICES

In some cases, the client may hold all the application state information, and the server holds none; it simply responds to each client request without keeping any details of the transaction history. This class of service is thus referred to as stateless; see Figure 5.19.

Figure 5.19 provides a comparison of stateful services, in which the server process keeps track of the state of the application, with stateless services, in which the server treats each new client request independently of all others and does not keep state information concerning the current application activity. The stateless approach is very powerful for achieving robustness, especially in high-scale systems. This is primarily because the failure of a stateful server disrupts the activities of all connected clients and requires the state be recovered to ensure ongoing consistency when the server is restarted, whereas the failure of a stateless server does not lose any state information. When a stateless server fails, its clients can be connected to a new instance of the server without the need for any state recovery (because each client maintains the state for its own application session, locally). Another way in which robustness is enhanced with the stateless server approach is that it leads to lower complexity in server components, making them easier to develop and test. The use-case multiplayer game provides an example of a stateful server design; the server stores the game state such as which player's turn it is to make a move.

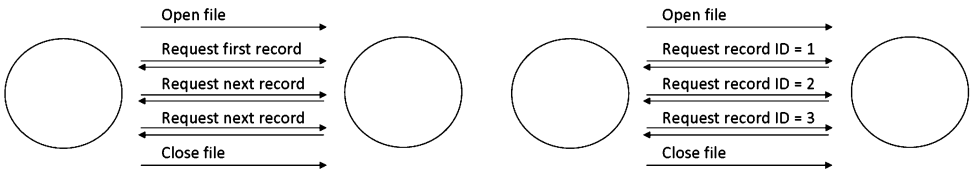


FIGURE 5.19

Stateful versus stateless services.

5.7.6 MODULAR AND HIERARCHICAL CS SYSTEMS

Having only two-component types may be limiting in some situations. There may be a need to divide the functionality of a service across several subcomponents to ensure an effective design and to limit the complexity of any single component. Such distribution also tends to improve robustness, maintainability, and scalability. Consider the situation where a company has its own in-house authentication system, which is intended to validate all service requests in several different applications. There are three different CS applications used to provide the services necessary for the company's business, and each user has a single set of access credentials to use all three services.

One option would be to integrate the authentication system into each of the business services, so that when any client request arrives (at any of the servers), the same authentication activity would occur. This approach is undesirable because it requires that all three applications are reengineered and that the server components grow in terms of complexity, actual code size, and run-time resource requirement. The authentication logic has to be duplicated across the three services and this costs greater design, development, and testing effort. This approach also means that any future upgrade of the authentication system must be performed on all three copies; otherwise, there will be differences in behavior and the authentication strength may become inconsistent.

A better option from an architectural viewpoint would be to develop the authentication system as a separate service (let us assume that the CS model is appropriate for the purpose of the example). The server side of the authentication system will actually perform the authentication checks, which will involve comparing the credentials provided by clients against data held by the service to determine who has the rights to perform various actions or access data resources. The client side of the authentication service can be "thin" (see discussion above), such that it is suitable for integration into other services without adding much complexity. The authentication client can thus be built into the three business services the company uses, such that the authentication is consistent regardless of which business service is used and so that the authentication service can be maintained and upgraded independently of the business services.

To illustrate this hierarchical service scenario, a specific application example is presented: consider an application comprising two levels of service (database access and authentication service). The system is built with three types of component: the database server (which consists of the database itself and the business logic needed to access and update the data), the database client (which provides the user-local interface to access the database), and an authentication server (which holds information necessary to authenticate users of the database). Note that the database server needs to play the role of a client of the authentication service as it makes requests to have its users authenticated. This provides an example where a single component changes between client and server roles dynamically or plays both roles simultaneously,[1] depending on the finer details of the behavioral design. This example is illustrated in Figure 5.20.

Figure 5.20 illustrates a modular approach to services in which a database server makes service requests to a separate authentication service. Both of these services are implemented using the CS model, and the authentication client-side functionality is embedded into the database server component. In this way, the database server effectively becomes a client of the authentication service. This approach has

[1] A human behavior analogy of the changing of component's role is provided by the relationship between a salesperson in a shop and the customer. The salesperson (the server) serves the customer (who is the client). However, the salesperson may need to order some special goods for the customer, from another distributor. Thus, the salesperson's role changes to customer for the purposes of the second transaction when the salesperson requests goods from the external supplier.

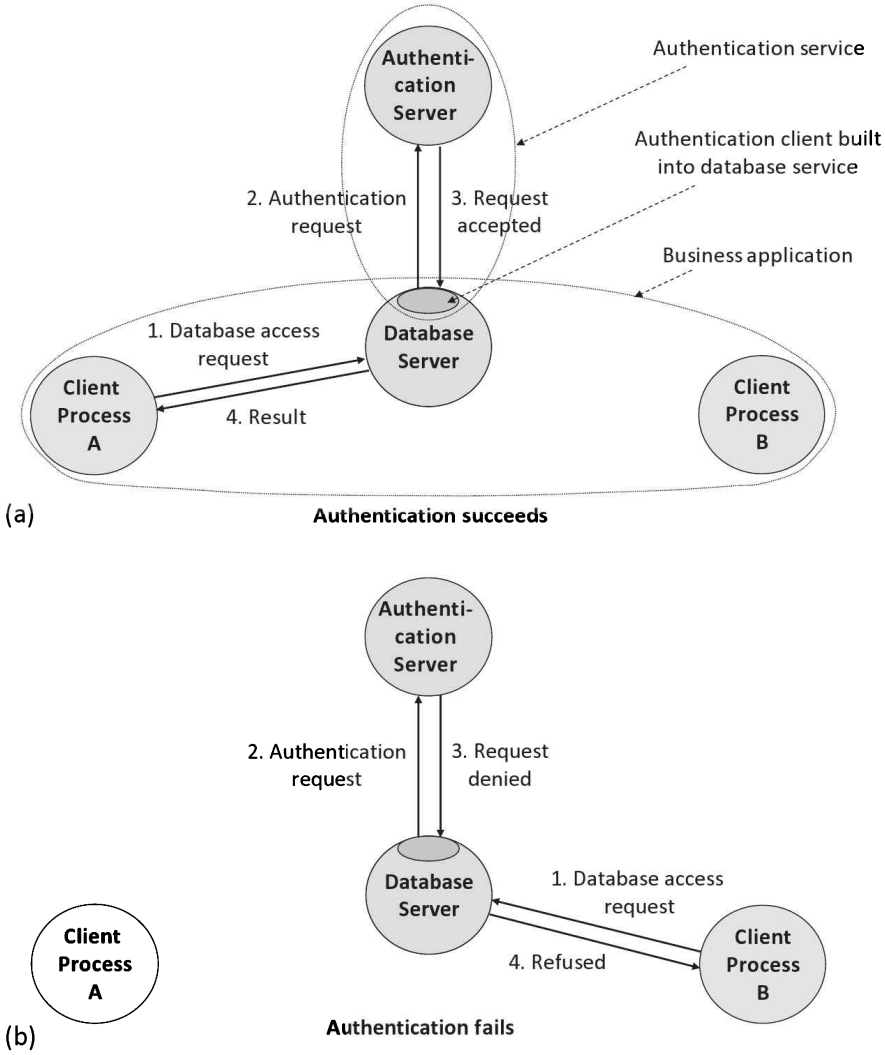


FIGURE 5.20

Modular services; the database service uses the authentication service.

further advantages in terms of transparency and security. The users do not need to access the authentication service directly; all accesses are performed indirectly via the other applications. This means that the user is not aware of the internal configuration and communications that occur. In part A of the figure, client A is authenticated to access the database, while in part B, client B is refused access; in both cases, it appears to the clients that the database server has performed the authentication check itself.

CS architectures can extend to situations in which a client needs access to more than one server, especially where an application component needs to access multiple remote or shared resources. An example of this is where the data required is spread across multiple databases or where a single database is itself distributed throughout a system. The scenario where a single component needs to access two separate databases is explored with the aid of Figure 5.21 in the next section.

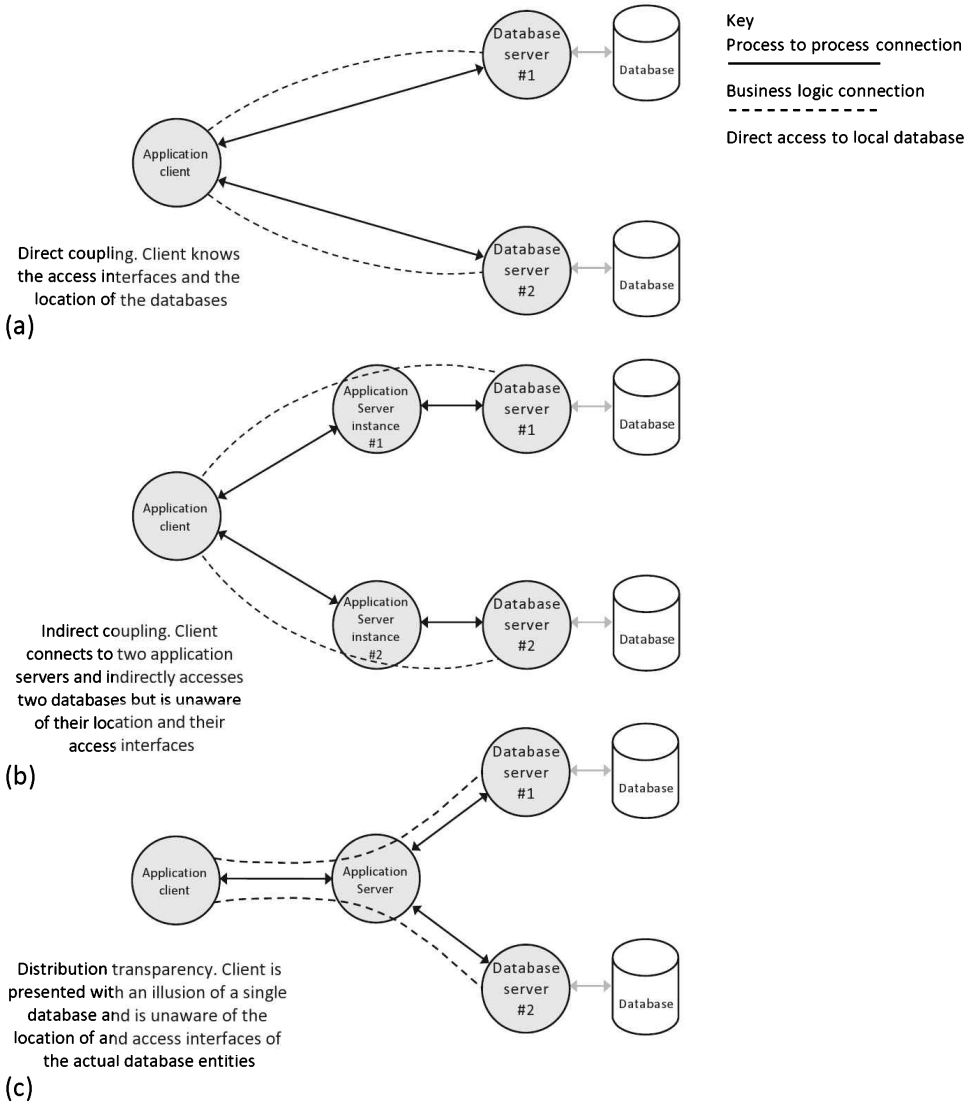


FIGURE 5.21

Two-tier direct coupling versus three-tier indirect coupling to services.

5.8 THREE-TIER AND MULTITIER ARCHITECTURES

CS is a form of two-tier architecture, its main architectural strength being its simplicity. However, it is not particularly scalable for several reasons, which include (1) the application logic concentrated in the single server component type (and thus the complexity of the component increases approximately linearly with functionality); (2) the flexibility limitations and robustness limitations that arise from the direct communication relationships between the service user and the service provider; and (3) the performance bottleneck that arises because all clients connect to a specific service, which may comprise a single process instance.

The introduction of additional tiers, that is, splitting the application logic into smaller modules and distributing it over more types of component, provides greater flexibility and scalability and can also have other benefits such as robustness and security. These architectures have three or more tiers and are commonly referred to as three-tier architectures, but the term multitier is also used. Note that neither usage is strictly adhered to in terms of the actual number of tiers implemented.

The benefits of the three-tier architecture are illustrated using an example for which it is well suited: online banking. Consider some of the main design requirements for such an application:

- The service must be secure. There must be no unauthorized access to data.
- The service must be robust. There must be no data corruption and the system must remain consistent at all times. Individual component failures should not lead to overall system failure, and users should not be able to detect such component failures.
- The service must be highly scalable. There should be a straightforward way to expand capacity as the bank's customer base, or the set of services offered, grows.
- The service must be flexible. The bank needs to be able to change or add functionality. The operation of the system itself needs to comply with legislation such as data protection and privacy laws, which can be changed from time to time.
- The service must be highly available. Customers should be able to gain access to the service at any time of the day or night (the target is a 24/7/365 service).

Although these requirements have been stated specifically in the context of an online banking scenario, it turns out that they are actually generally representative of the requirements of a large proportion of corporate distributed applications. They are listed in a possible order of priority for a banking application, and this order will not necessarily be the same for other applications. In essence, the requirements are security, robustness, scalability, flexibility, and availability. Keeping the example quite high level, let us now consider the ways in which the three-tier architecture satisfies or contributes to each of these requirements.

Security. The client is decoupled from the data access tier by the middle business logic tier. This allows for robust user validation to occur before user requests are passed to the security-sensitive components. It also obscures the internal architecture of the system; an attacker may be able to send a fake request to the middle tier, but the system can be set up such that the third tier only accepts requests from the middle tier and that, when viewed externally from the bank's network, the third tier is not detectable.

Robustness. The three-tier architecture enables replication to be implemented at all tiers except the first tier. This is because each user must have exactly one client process to access the system; there can be many clients active at once, but this is not replication as each client is unique and is controlled independently by its user. The business logic tier and the data access tier can each be replicated, to different extents as necessary, to ensure that the service is robust. It is the design of the replication mechanism at the data access tier that contributes most significantly to ensuring data consistency despite individual component failures.

Scalability. Replication at both the business logic and the data access tiers also contributes to scalability. Additional instances of the components can be added to meet growing service demand. The second and third tiers can be scaled up asymmetrically depending on where the bottlenecks occur. For example, if user validation becomes a choke point as the load on the validation mechanisms is increased, then the business logic layer can be expanded with a higher replication factor than the data access layer, which may continue to perform satisfactorily with its current level of resource.

Flexibility. Recall that three-tier architectures are also sometimes called multitier. This is because there does not have to be exactly three layers and the use of terminology is not always precise. Consider what would happen if not only the user validation mechanism were to become a serious bottleneck due to increased load (as described in the paragraph above) but also the legislation governing the way that banks perform user validation was tightened up requiring significantly stronger checks are put in place (which are correspondingly more resource-intensive). In such a case, if the validation remains part of the business logic, this component type will become very complex and heavyweight. A better approach could be to introduce a new layer into the system and to separate out the user validation logic from the other business logic. The multitier approach allows for changes like this to occur in one layer without affecting the other components, so, for example, the user-client component and the data access component can ideally remain unchanged. This limits the cost of change and perhaps more significantly reduces the risks of instability that arise if too many things are changed at once.

Availability. Once again, it is the flexibility of being able to replicate at multiple layers that provides the basis for a highly available service. If some of the replicas are located at different physical sites, then the service can even continue to operate despite site-local disasters such as floods and power cuts. As long as at least one component of each type is operating, then it is possible that the overall service will be fully functional. Note however that this must be subject to meeting all other requirements; for example, the data consistency aspect of the robustness requirement may enforce that a minimum of, for example, three data access components are running at any time.

Three-tier architectures have further advantages over two-tier designs, the most important of these being the greater extent of transparency they provide and the additional benefits associated with this.

A generic database-oriented application that accesses two different databases is used to exemplify the transparency benefits arising through progression from a two-tier design to a three-tier design; this is explained with the aid of Figure 5.21.

Figure 5.21 illustrates the benefits of using additional tiers. The figure shows three possible ways to connect an application component to a pair of databases. Part (a) of the figure shows a two-tier configuration in which the component is connected directly to the database services. In this situation, all business logic is implemented in the single application component (which is shown as a client in the figure, because it makes service requests to the database service). This configuration is an example of direct coupling and has the advantage of low initial development cost and operational simplicity.

However, this configuration lacks transparency in several important ways. This configuration requires that the client-side application developer deals with the distribution of the components directly; we can say it is not distribution-transparent. The developer has to take account of which data are located at each database when developing the application code that accesses the data. Complex scenarios such as what should happen when only one of the databases is available, or if one fails to perform a transaction but the other succeeds (which can be especially complex in terms of data consistency), must be decided and the scenarios automatically detected and supported in the application logic.

In terms of location and access transparency, the client component must know or find the location of the database servers in order to connect and, because the connection is direct, must know the native format of the database interface. If the database type were changed, then the client component may have to be updated to account for changes in the interface to the databases, such as the message formats used, and the way in which logical connectivity with the database server is achieved (e.g., the way in which the client authenticates itself may change or be different for each of the two databases).

Part (b) of Figure 5.21 shows a more sophisticated alternative architecture in which a third tier is introduced. Part of the business logic is moved into application-level server components that handle the connectivity to and communication with the database services.

The application-level servers can perform additional functions such as security and access control, providing access and location transparency to the client: the former in the sense that the databases may have their own native formats, which the application server deals with, thus hiding these differences from the client, and the latter because the application server can deal with locating the database service without the client needing to know its location or having the required mechanism to locate it.

This approach is superior in several ways. Firstly, it hides the possible heterogeneity of the databases from the client making the client logic simpler. Secondly, it decouples the client component from the database components potentially improving security and robustness since authentication and filtering of requests can be added into the new tier if necessary, and thus, the client cannot directly manipulate the data. Thirdly, many clients can potentially use the same application server component. This centralizes the application business logic making the application generally easier to maintain and simplifying business logic upgrades.

Part (c) of Figure 5.21 shows a further developed architecture in which an application-level server deals with all aspects of database connectivity, such that the client is presented with the illusion that there is only a single database entity. This is achieved because the client only connects to a single application server, which takes care of the connection to the multiple databases, providing the client with a single-system abstraction. In this configuration, the server provides distribution transparency in addition to the other forms of transparency achieved by configuration (b).

Notice that configuration (c) is an example of a three-tier application: The application client is the first tier (the user interface), the application server is the second tier (the business logic), and the database server is the third tier (managing the data access). However, this arrangement of components could also be considered to be effectively two levels of CS in which the application logic is developed as a CS application, and the database is itself developed as a self-contained CS application and the two then merged together. In fact, this might be a helpful way to reason about the required functionality of the various components during the design phase. In the resulting three-tier architecture, the middle component (the application server) takes the role of client in one CS system and has the role of server in the other; see part (b) of Figure 5.22.

Figure 5.22 illustrates some mechanistic aspects of the three-tier architecture. Part (a) of the figure shows a possible communication configuration using sockets. In the configuration shown, the middle tier connects to the data tier (the data tier is the passive side and the middle tier is the active side that initiates the connection). The application client connects to the middle tier whenever service is needed (the application client is the active side and the middle tier is the passive side in this case). Part (b) of the figure shows a possible way to construct a three-tier architecture as two CS stages. This way of visualizing three-tier systems is particularly useful when considering the extension of existing two-tier applications or during the design phase of a three-tier application because it facilitates a modular approach to the design, and in particular, it can help with the elicitation of clear design requirements for the various components. The mechanistic detail shown in Figure 5.22 is applicable to both architectures (b) and (c) depicted in Figure 5.21.

The multiple database scenarios described above have been developed as a series of three application versions, which each map directly onto the configurations illustrated in Figure 5.21. Activity A1 explores the behavior of the three versions, to facilitate understanding of aspects such as the transparency differences between the configurations, the relative strengths and weaknesses of the different configurations, and the different coupling approaches represented.

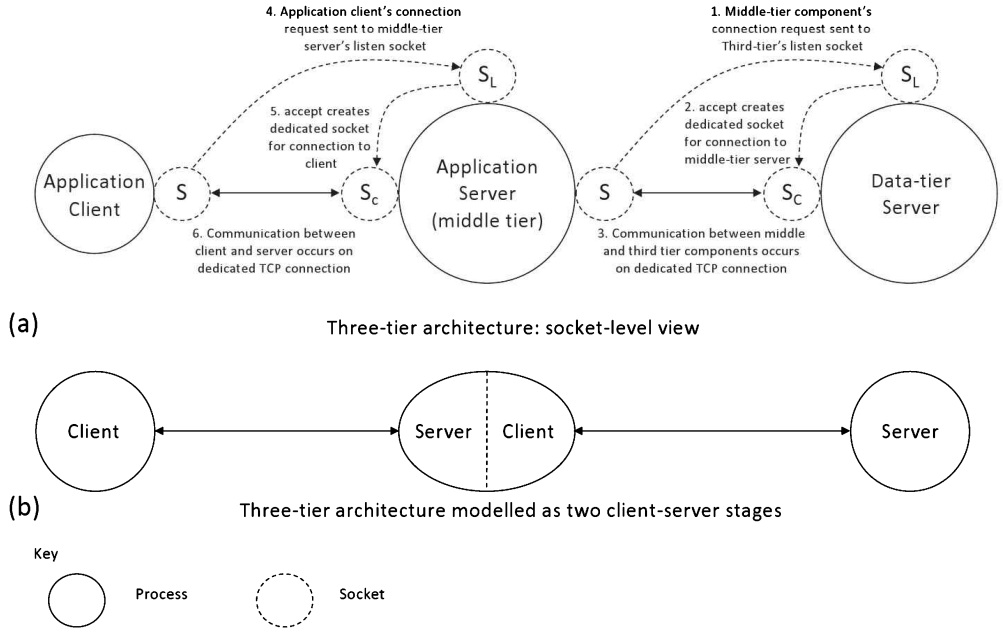


FIGURE 5.22

The three-tier architecture; some mechanistic aspects.

The components in the three applications communicate over TCP connections, configured in the same manner as shown in part (a) of Figure 5.22.

There can be many reasons why data are distributed within a system. Data may be held in a variety of places based on, for example, the type of data or the ownership of the data or the security sensitivity of the data. It may be purposefully split across different servers to reduce the load on each server, thus improving performance. It may also be replicated, such that there are multiple copies of the same data available in the system. The distribution of data using different physical databases may also be a design preference to ensure maintainability and to manage complexity. For example, a retail company will very likely have an information structure that separates out the customer details into a customer database, the stock details into a stock database and the staff, and payroll details into a further database.

For any large system, the data are much more likely to be distributed than not. It would be generally undesirable for a large organization to collect all of its data about customers, suppliers, products, etc., into a single database, which would be complex and unwieldy, possibly require a very large amount of localized storage, and likely be a performance bottleneck because all data queries would be passed to the same server. In addition, many e-commerce applications operate across multiple organizations, and they are very unlikely to allow their data to be stored remotely, at the other organization's site. More likely, each organization will split their data into two categories: that that they are happy to allow remote organizations to access via a distributed e-commerce application and that that they wish to keep private.

The designers of distributed applications are thus faced with the challenge of accessing multiple databases (and other resources) from within a specific application such that the users of the system are unaware of the true locations of the data (or other resources) and do not have to be concerned with the complexity of the physical configuration of the underlying systems. This is a very useful

ACTIVITY A1 EXPLORING TWO-TIER AND THREE-TIER ARCHITECTURES

Three differently structured versions of the same application are used to explore a number of important aspects of software architectures.

The applications each access two different remote database servers. To simplify the configuration of the software and to ensure that the focus of the experimentation is on the architectural aspects (and not the installation and configuration of databases), the database server programs used in the activity actually hold their data in the form of in-memory data tables instead of real databases. This does not affect the architectural, communication, and connectivity aspects of their behavior.

Prerequisites

Copy the support materials required by this activity onto your computer; see Activity I1 in Chapter 1.

Learning Outcomes

To gain an understanding of different configurations of CS

To gain an appreciation of the distributed nature of resources in most systems

To understand robust design in which some services may be absent, but the remainder of the system still provides partial functionality

To understand the differences between two-tier and three-tier architectures

To gain an appreciation of different coupling approaches

To explore access transparency in a practical application setting

To explore location transparency in a practical application setting

To explore distribution transparency in a practical application setting

The activity is performed in three main stages.

Method Part A: Understanding the Concept of Distributed Data

The first part uses a two-tier implementation comprising the following software components: DB_Arch_DBServer_DB1_Direct (this is the implementation of the first of the two database servers and is used throughout the entire activity), DB_Arch_DBServer_DB2_Direct (this is the implementation of the second database server and is also used throughout the entire activity), and DB_Arch_AppClient (this version of the client connects directly to each of the two database servers). This configuration corresponds to part (a) of Figure 5.21.

The data needed by the client are split across the two database servers, such that each can provide a specific part of it independently of the other. One database server holds customer name details; the other holds customer account balance details. The single client thus needs to connect to both servers to retrieve all the relevant data for a particular customer but can retrieve partial data if only one of the servers is available.

This part of the activity demonstrates direct connection between the application client and the database servers. The location (address and port) of the database servers has to be known by the client. In the demonstration application, the port numbers have been hard-coded, and the server IP address of each server is assumed to be the same as that of the client; the user must enter the correct IP address if the server is located on a different computer.

The connectivity and behavior can be explored in three substeps.

Part A1. Run the client with a single database server.

Start by running the client (DB_Arch_AppClient) and the first of the two database servers (DB_Arch_DBServer_DB1_Direct) on the same computer. Attempt to connect the client to each server. This should succeed for the first database server and fail for the second. Since each connection is managed separately by the client in this particular design, the absence of one server does not interfere with the connectivity to the other.

Now request some data. Provide a customer ID number from the set supported {101, 102, 103} and click the "request data" button. You should see that the partial data held by the particular server that is running are retrieved. The application is designed to send the same request (i.e., the key is the customer ID) to all connected database servers, so the fact that one of the servers is unavailable does not prevent the other from returning its result, that is, only the customer name is returned to the client.

Part A2. Run the client with the other database server.

Confirm that that behavior is the symmetrical with regard to whichever of the databases is unavailable. This time, run the client (DB_Arch_AppClient) and the second of the two database servers (DB_Arch_DBServer_DB2_Direct) on the same computer. Follow the same steps as before. Once connected and a customer data request has been submitted,

ACTIVITY A1 EXPLORING TWO-TIER AND THREE-TIER ARCHITECTURES—Cont'd

the database should return the appropriate customer account data, even though database #1 is not available to supply the customer name.

Part A3. Run the client with both database servers.

Now run the client and both database servers on the same computer. Connect the client to each database server, and then, submit a data request. Notice that the request in this case is sent to both database servers and that both respond with the data they hold, based on the customer ID key in the request message.

You may be able to just about notice the delay between the two data fields being updated in the client, which arises because the data arrive in two messages, from the two different database servers.

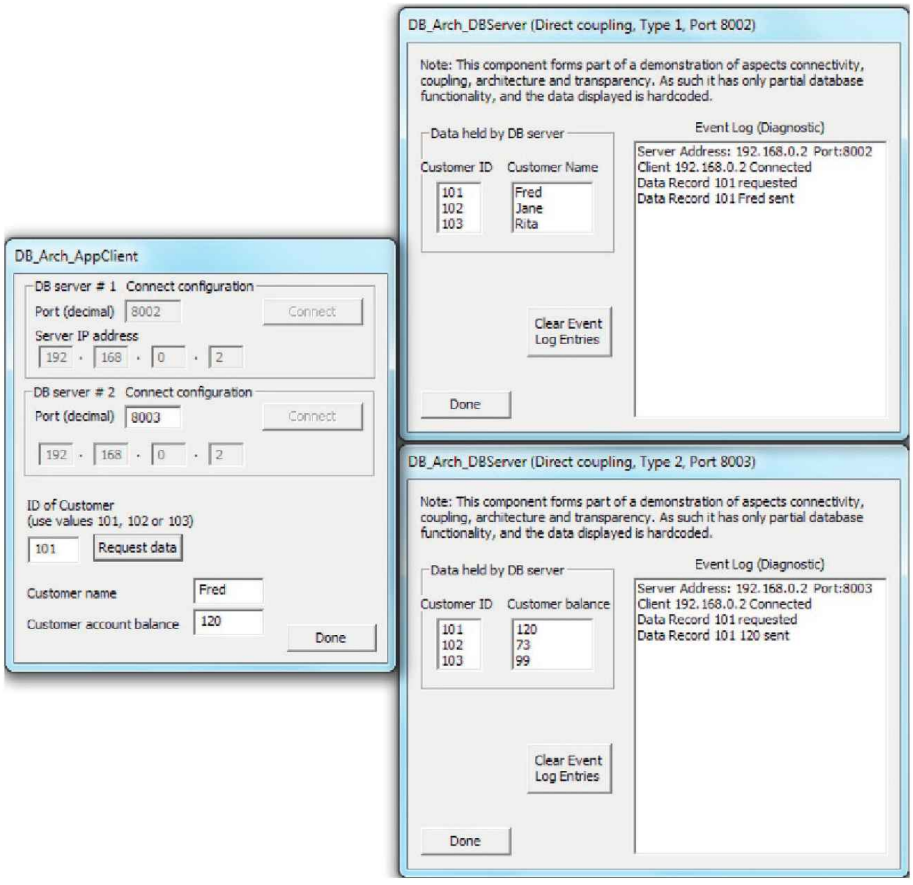
Expected Outcome for Part A

You should see that the distributed application functions correctly and that its behavior corresponds to what is expected based on the procedure described above.

You should see how the client sets up and manages the connections to the two databases separately.

Use the diagnostic event logs provided in the database server components to see details of the connections being established and messages being passed between the client and the servers. Make sure that you understand the behavior that occurs.

The screenshots below show the resulting behavior when all three components interact successfully, for part A3 of the activity.



ACTIVITY A1 EXPLORING TWO-TIER AND THREE-TIER ARCHITECTURES—Cont'd

Method Part B: Extending to Three Tiers

This part of the activity is concerned with the introduction of a middle tier to decouple the application client from the database servers (which were directly connected to the client in the configuration used in part A).

This second part of the activity introduces a pair of application servers that each deal with the connection to one of the database servers. This configuration comprises the following five software components: DB_Arch_DBServer_DB1_Direct (the implementation of the first of the two database servers), DB_Arch_DBServer_DB2_Direct (the implementation of the second of the two database servers), DB_Arch_AppServer_for_DB1 (the middle tier that connects to the first database type), DB_Arch_AppServer_for_DB2 (the middle tier that connects to the second database type), and DB_Arch_AppClient_for_AppServer1and2 (a modified client component that connects to the new middle-tier application servers instead of directly connecting to the database servers). This configuration corresponds to part (b) of Figure 5.21.

The important difference from the configuration in part A is that now, the client only knows its application servers. It still has to connect to two other components, but it is now indirectly coupled to the data access tier and does not need to know how that tier is organized (in the configuration used in part A, it was directly coupled).

The connectivity and behavior are explored in three substeps.

Part B1. Run the client, the application server that connects to DB1 and the DB1 database server. Request data for one of the customer IDs and observe the result. Notice that the client still manages its connections separately, and thus, data can be received from DB1 in the absence of the other database.

Part B2. Now, run the client with the other application server (which connects to DB2) and the DB2 database server. Confirm that the client's behavior is symmetrical with regard to the unavailable components and that data can be received from DB2 in the absence of DB1.

Part B3. Run all five components such that the complete three-tier system is operating. Connect each application server to its respective database server, connect the client to the application servers, and then submit a data request. Notice that the request is sent separately to each application server, which then forwards it to its database server. Note also that the responses from the database servers are forwarded on the client, which displays the resulting data.

As with part A3, you may also be able to just about notice the delay between the two data fields being updated in the client. It may be slightly more obvious in this case, as the two sets of messages have to pass through longer chains of components, increasing latency.

Expected Outcome for Part B

By experimenting with this configuration, you should be able to see some of the differences that arise as a result of this being a three-tier application, in which the application client is the first tier (user interface), the application server is the second tier (business logic), and the database server is the third tier (data access).

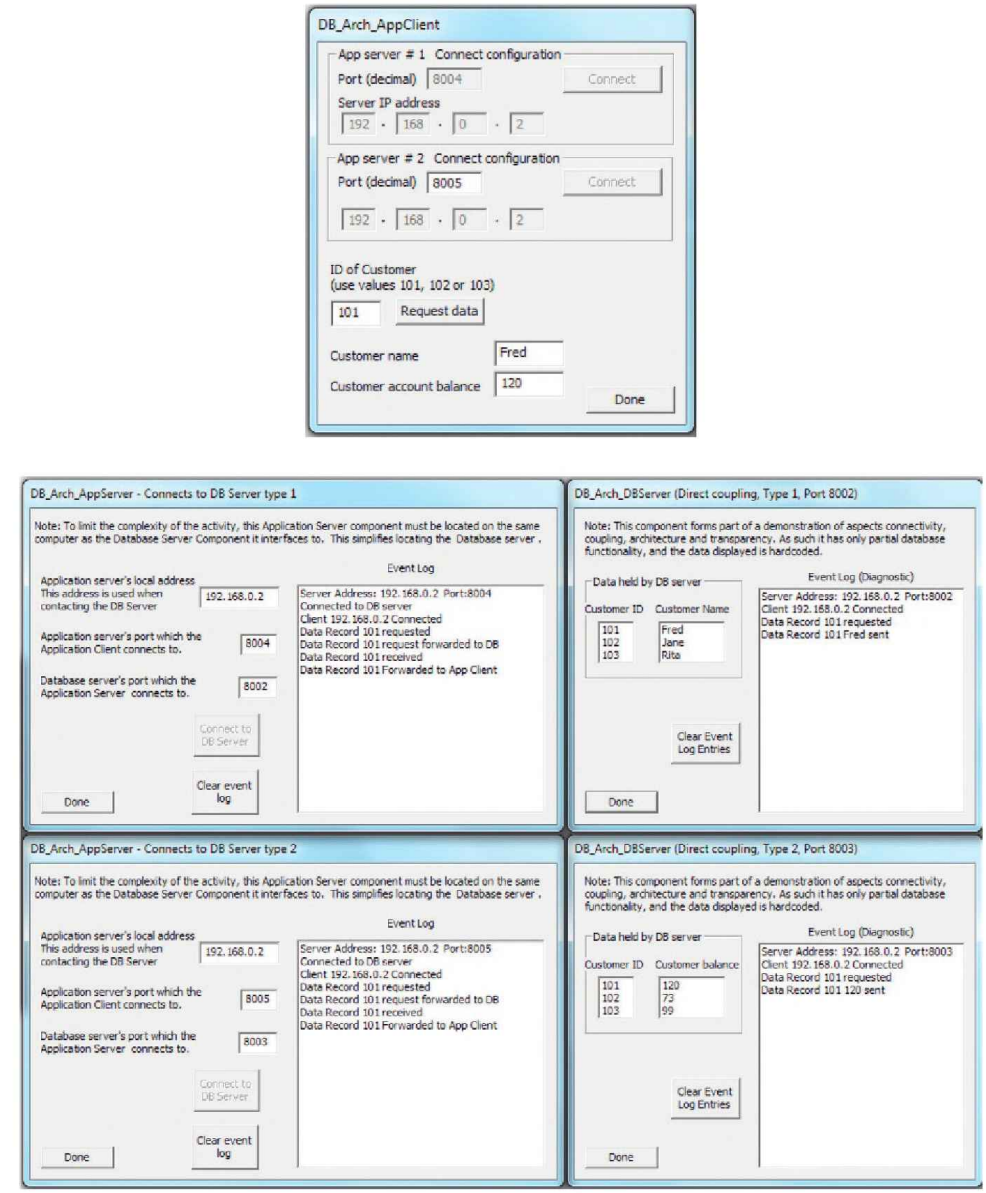
Use the diagnostic event logs provided in the application server components and the database server components to see details of the connections being established and messages being passed between the three tiers. Make sure that you understand the behavior that occurs.

The application servers break the direct coupling that was used in the two-tier architecture. The application servers provide location and access transparency. Location transparency is achieved because the application servers connect to the database servers such that the client does not need to know the location details of them and does not even need to know that the database servers are external components. Access transparency is achieved because the application servers hide the heterogeneity of the two databases, making all communication with the client homogeneous, using a consistent application-specific communication protocol.

It is also important to note that the demonstration has limited scope and that in a real application, the middle tier could additionally provide services as authentication of clients and of the requests they make, before passing them on to the database service.

The screenshots below show the resulting behavior when all five components interact correctly, for part B3 of the activity.

ACTIVITY A1 EXPLORING TWO-TIER AND THREE-TIER ARCHITECTURES—Cont'd



ACTIVITY A1 EXPLORING TWO-TIER AND THREE-TIER ARCHITECTURES—Cont'd

Method Part C: Achieving Distribution Transparency

The third part of this activity is concerned with achieving the transparency goals of distributed systems; in this particular case, we focus on hiding the details of the database services and connectivity such that the client is provided with the illusion that there is a single, locally connected database. This is achieved by refining the middle tier such that the client only has to connect to a single application server that takes care of the connection to the separate databases.

The third configuration uses a modification of the three-tier implementation used in part B. This comprises the following four software components:

DB_Arch_DBServer_DB1_Direct (the implementation of the first of the two database servers), DB_Arch_DBServer_DB2_Direct (the implementation of the second of the two database servers), DB_Arch_AppServer_for_DB1andDB2 (the middle tier that connects to the both of the databases, transparently from the viewpoint of the client), and DB_Arch_AppClient_for_SingleAppServer_DB1andDB2 (a modified client component that connects to the single new middle-tier application server). This configuration corresponds to part (c) of Figure 5.21.

The important difference from the configuration in part B is that now, the client only knows its single application server. It has no knowledge of the way in which the data are organized and cannot tell that the data are retrieved from two different databases. The client is indirectly coupled to the database servers.

The connectivity and behavior are explored in three substeps.

Part C1. Run the client, the application server and the DB1 database server. Request data for one of the customer IDs and observe the result. Notice that the client sends a single request to the application server (the client is unaware of the database configuration). This time, it is the application server that manages the connections to the two database servers, and for robustness, it handles them independently. Therefore, the application server is able to return partial data to the client when only one of the databases is connected.

Part C2. Repeat part C1, this time with only the other database server available.

Part C3. Run the system with all components connected correctly. Request data for one of the customer IDs and observe the result. The single request sent by the client is used by the application server to create a pair of tailored requests, one to each specific database. Responses returned by the two databases are forwarded on by the application server as two separate responses to the client.

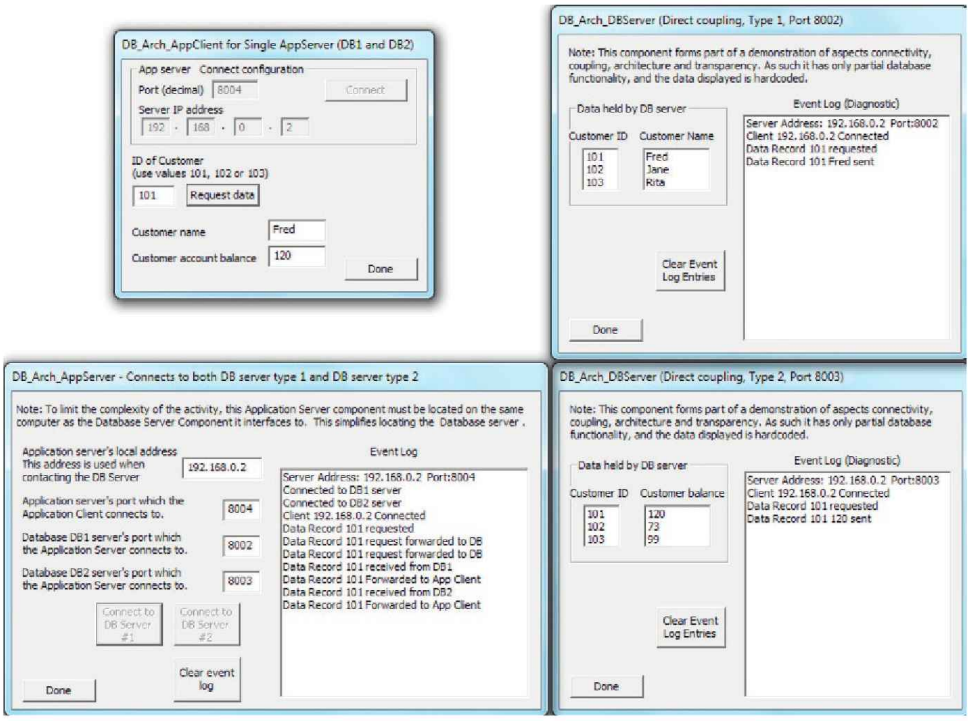
Expected Outcome for Part C

Through the experiments in this part of the activity, you should be able to understand the way in which distribution transparency has been added, where it was absent in part B. The client is unaware of the presence of the two different databases and their interfaces. The client is presented with a single-system view in the sense that it connects to the single application server component that provides it all the resources it needs; the design of the client is not concerned with the configuration of the rest of the system.

As with the other parts of the activity, use the diagnostic event logs provided in the application server and the database server components to inspect the behavior in detail.

The screenshots below show the resulting behavior when all four components interact successfully, for part C3 of the activity.

ACTIVITY A1 EXPLORING TWO-TIER AND THREE-TIER ARCHITECTURES—Cont'd



Reflection

This activity provides some insight into the design and operation of two-tier and three-tier applications.

The two-tier configuration used in part A lacks transparency because the client directly connects with the two database servers. The second configuration introduces the third tier, but in such a way as to limit the extent of transparency achieved (access and location transparency are provided). The third configuration reorganizes the middle tier to add distribution transparency such that the client process is shielded from the details of the database organization and is not even aware that there are multiple databases present.

A good exercise to reinforce your understanding of the concepts shown in this activity is to choose any distributed application (some examples are banking, online shopping, and e-commerce) and to sketch outline designs for it, using both the two-tier and the three-tier architectures. Evaluate the designs in terms of their expected transparency, scalability, and flexibility.

example of the importance of transparency. Care has been taken to ensure that the applications developed to support Activity A1 capture both the challenge of accessing multiple distributed resources and the resulting transparency that can be achieved through appropriate design. Note that the actual database functionality that has been implemented in the applications explored in Activity A1 is minimal; it is just sufficient for the examples to be understandable, as the learning outcomes of the activity are focused around the communications and architectural aspects rather than the database itself.

5.9 PEER-TO-PEER

The term peer means "with equal standing"; it is often used to describe the relative status of people, so, for example, if you are a student in a particular class, then your classmates are your peers. In software, peers are components that are the same in terms of functionality. The description peer-to-peer suggests that for such an application to reach full functionality, there need to be multiple peer components interacting, although this is dependent on the design of the actual application and the peer components. To place this in context, if the purpose of the application is to facilitate file sharing, then each peer component may have some locally held files that it can share with other peers when connected. If a user runs one instance of the application in the absence of other peers, then only the locally held files will be available; this may still be useful to the user. In contrast to this, consider a peer-to-peer travel information service in which users share local travel information such as news of train delays or traffic jams with other users of the application. The locally held information is already known to the local user (because they create it), so this particular application is only useful when multiple peer components are connected.

5.9.1 CHARACTERISTICS OF PEER-TO-PEER APPLICATIONS

The essence of a peer-to-peer application is that two or more peers connect together to provide a service to the users. Some peer-to-peer applications are designed to work well with just a small number of users (such as media sharing), while for some applications, the utility increases in line with the number of peers connected; for example, some games become increasingly interesting when the number of players reaches a critical mass.

Mechanistically, if peers are symmetrical, then any instance can offer services to the others. This means that in a typical peer-to-peer application, any component can be a service provider or a service requester, at different times depending on circumstances. This is a significant difference from the CS and three-tier models, in which the various components have predetermined roles that are not reversible, for example, a client and server cannot swap roles because the specific behaviors are embedded into the different logic of each component type.

From an interaction viewpoint, peer-to-peer applications tend to rely on automatic discovery and connection among peers. Interactions are sometimes "by chance" due to coexistence of users with related needs. For example, peer-to-peer is popular with mobile devices for game playing and file/media sharing, using wireless links such as Bluetooth for connection. If such applications are enabled on devices, then a connection will be automatically established when the devices are in close proximity to one another. Since it is not generally predictable when peers will come into contact and establish relationships, peer-to-peer applications are often described as having ad hoc interaction behavior. This is in contrast with the CS model, in which the interaction is quite structured and in some senses choreographed at design time (in the sense that a particular client is designed to connect to a particular server, either automatically upon start-up or when the user explicitly requests).

The general characteristics of peer-to-peer applications can be summarized as follows:

- Peers communicate with others to achieve their function (e.g., games, messaging, and file sharing).
- The applications often have limited scope (typically with a single main function) and the requirement for connection to remote "others" on a simple and flexible basis.
- Connectivity is ad hoc (i.e., it can be spontaneous, unplanned, and unstructured).
- Peers can interact with others in any order, at any time. Figure 5.23 captures the essence of this.
- Peer-to-peer is well suited to mobile applications on mobile devices.

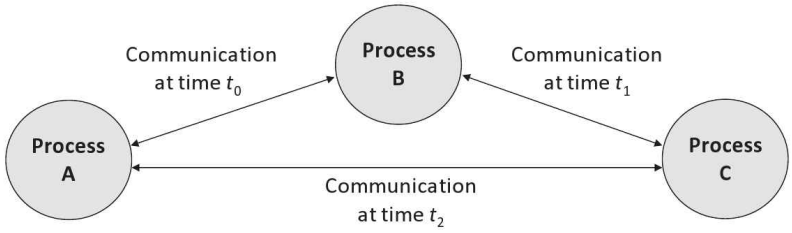


FIGURE 5.23

Peers communicate with different neighbors at different times depending on the application requirements and peer availability.

Figure 5.23 illustrates the dynamic nature of peer-to-peer applications, in which peers may join or leave independently at various times. Therefore, a group of peers may not all be present or connect to each other at the same time; the connections may occur opportunistically as different peers become available.

5.9.2 COMPLEXITY OF PEER-TO-PEER CONNECTIVITY

The complexity of peer-to-peer connectivity is potentially much higher than in other architectures, which are more structured. In CS, the number of connections to a given server is one per client, so if there are n clients, there are n connections to the server; thus, we say the interaction complexity is "order n" stated O(n). This is a linear relationship. With this in mind, consider a peer-to-peer scenario where five peers are present and there are multiple connections established in an ad hoc manner among them.[2] A possible outcome is shown in Figure 5.24.

Figure 5.24 shows a configuration of five peers with a total of seven connections between the peers. The maximum number of connections that would occur if all peers connected to each other peer would be ten, in which case each peer would have four connections to other peers. There is a calculation that can be used to determine the maximum number of connections that can occur between a certain number of peers, given in formula (5.1):

$$C = \frac{P(P-1)}{2} \tag{5.1}$$

where P is the number of peers present and C is the resulting maximum number of connections. Let's insert some values: for four peers, we get $C=(4*3)/2=6$; for five peers, we get $C=(5*4)/2=10$; while for six peers, we get $C=(6*5)/2=15$, and if we consider ten peers, we get $C=(10*9)/2=45$. This is clearly increasing in a steeper-than-linear fashion. Such a pattern of increase is generally described

[2]The phrase "connections established in an ad hoc manner" essentially means that there will be various connections established between a particular peer and its neighbors, depending on perhaps the sequence with which specific peers joined the group (because some applications may limit the number of peers that an individual connects to), and the actual range between each pair of individuals (so different subsets are visible to each individual). Therefore, the exact mapping of connections might not be predictable and may be different even in similar (slightly different) situations.

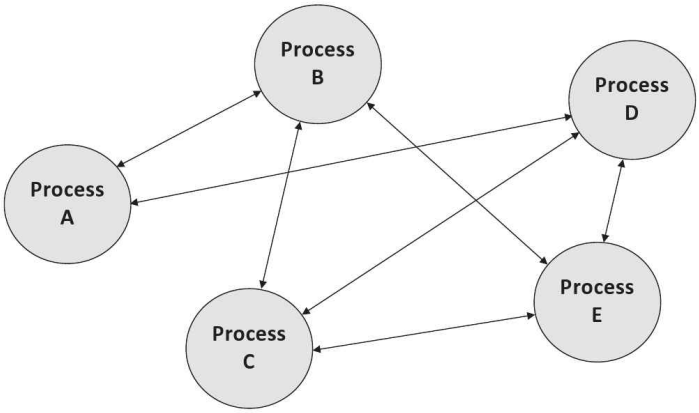


FIGURE 5.24

Ad hoc connectivity in peer-to-peer architectures.

as exponential, and in this particular case, it has order $O((n(n-1))/2)$, which is derived from formula (5.1). With such a steeply increasing pattern of communication intensity, at some point, the number of connections and the associated communication overheads will impact on the performance of the application, thus limiting scalability. In other words, there is a limit to the scale at which the system can grow to and still operate correctly and responsively.

However, some peer-to-peer systems are designed such that peers connect only to a subset of neighbors (i.e., those other peers that are in communication range). For example, some applications (including some sensor network applications) rely on peer connectivity to form a chain to pass information across a system.

5.9.3 EXPLORING PEER-TO-PEER BEHAVIOR

As mentioned above, a popular application for peer-to-peer architectures is media sharing (such as photos and songs) and is especially popular with mobile platforms such as phones and tablets.

A media-sharing peer-to-peer application MediaShare_Peer has been developed to facilitate practical exploration of a peer-to-peer application and is used in Activity A2. The exploration includes aspects of the way in which peer-to-peer systems work, their benefits in terms of flexibility and ad hoc connectivity, and the way in which automatic configuration can be achieved.

ACTIVITY A2 EXPLORATION OF THE PEER-TO-PEER ARCHITECTURE

A media-sharing application is used to explore a number of important aspects of the peer-to-peer architecture.

The application is based on the concept that each user has some music files that they are prepared to share with other users, on a like-for-like basis. The user runs a single peer instance. When there are no other peers available, the user can play only the songs that are held by the local instance. When another peer is detected, the two peers automatically discover each other (in terms of their addresses) and thus form an association (the term connection is

ACTIVITY A2 EXPLORATION OF THE PEER-TO-PEER ARCHITECTURE—Cont'd

avoided because it could imply the use of a connection-oriented transport protocol, such as TCP, when in fact this application uses UDP, partly because of the ability of UDP to broadcast, which is necessary for the automatic peer discovery). The peers then exchange details of the media files they each hold. At this point, the user display is updated to reflect the wider set of resources available to the user (the user can now play any of the music files held on either of the peers).

The demonstration application displays various diagnostic information including an event log, which indicates what is happening behind the scenes, such as the discovery of other peers, and also indicates which specific peer has supplied each music file. This information is included specifically to enhance the learning outcomes of the activity; in a real application, there is no need for the user to see such information.

To avoid unnecessary complexity in the demonstration application, only the resource file names are transferred (i.e., a list of music file filenames) and not the actual music files themselves. In a real implementation, when a user wishes to play a song, the actual song data file would need to be transferred if not already held locally. This simplification does not affect the value of the activity since it is focused on demonstrating the peer discovery and transparency aspects.

Prerequisites

Copy the support materials required by this activity onto your computer; see Activity I1 in Chapter 1.

Learning Outcomes

To gain an understanding of the peer-to-peer architecture
To become familiar with a specific example peer-to-peer application
To appreciate the need for dynamic and automatic peer discovery
To understand one technique for achieving automatic peer discovery
To appreciate the need for transparency in distributed applications
To explore the transparency provision in the demonstration application
The activity is performed in three main stages.

Method Part A: Running a Peer in Isolation

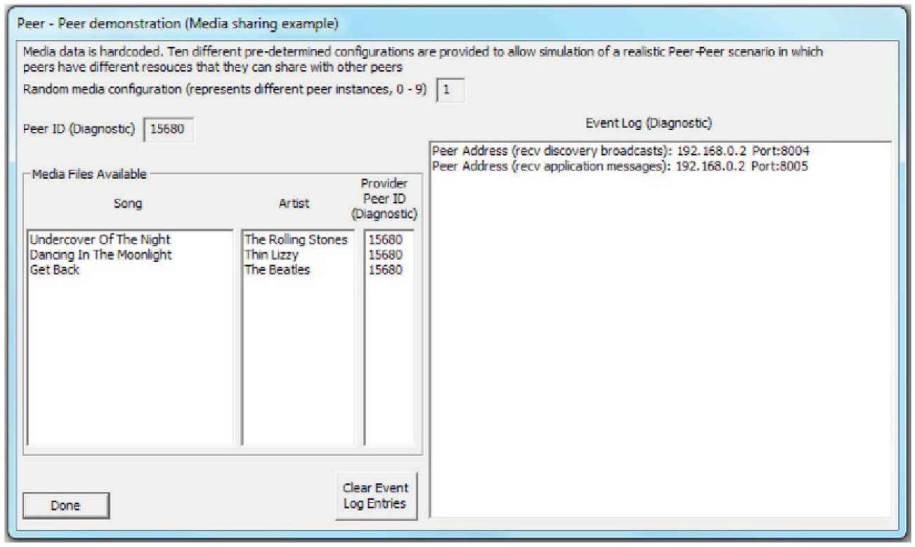
This part of the activity involves running a single instance of the MediaShare_Peer program on a single computer. Note the peer ID number, the list of songs and artists displayed, and the diagnostic event log entries. Close the peer instance and start it again, several times, each time noting the data displayed. You will see that the demonstration application has been designed to generate a unique ID randomly and also to select its local music files randomly, so that the single program can be used to simulate numerous peers with different configurations, without the user having to manually configure it.

Expected Outcome for Part A

This step provides an opportunity to familiarize yourself with the application in the simplest scenario. It also demonstrates two important concepts: Firstly, a single peer operates correctly in isolation (i.e., it does not fail or behave strangely when no other peers are available, but instead operates correctly as a peer-to-peer application that just happens to have only one peer). Secondly, a user sees a list of resources (song files in this case) available for use. Depending on the actual application requirements, it will be generally desired that locally held resources are always available to the local user regardless of the wider configuration of the other peers.

The screenshot below shows a single peer operating in isolation. The media files listed as available are the locally held resources, and thus, a user can access these (i.e., play the soundtracks) without needing any additional components.

ACTIVITY A2 EXPLORATION OF THE PEER-TO-PEER ARCHITECTURE—Cont'd



Method Part B: Automatic Peer Discovery

Leaving the local peer instance running (from part A), start another instance on another computer in the same local network (this is important because broadcast communication is used, which is blocked by routers). You can also confirm empirically that only one peer instance can be operated on each computer (try running two copies on one computer). This restriction arises because each peer has to bind to both of the application ports that are used (one for receiving peer self-advertisement messages for automatic peer discovery and one for receiving the peer-to-peer application messages).

Discovery is achieved by each peer sending a self-advertisement message (PEER_SELF_ADVERTISEMENT message type) containing the unique ID of the sending peer, as well as the IP address and port number details that should be used when application messages are sent to that peer. On receipt of these messages, a peer stores the details in its own, locally held known_peers array. This enables the recipient of future self-advertisement messages to distinguish between peers already known to it and new peers.

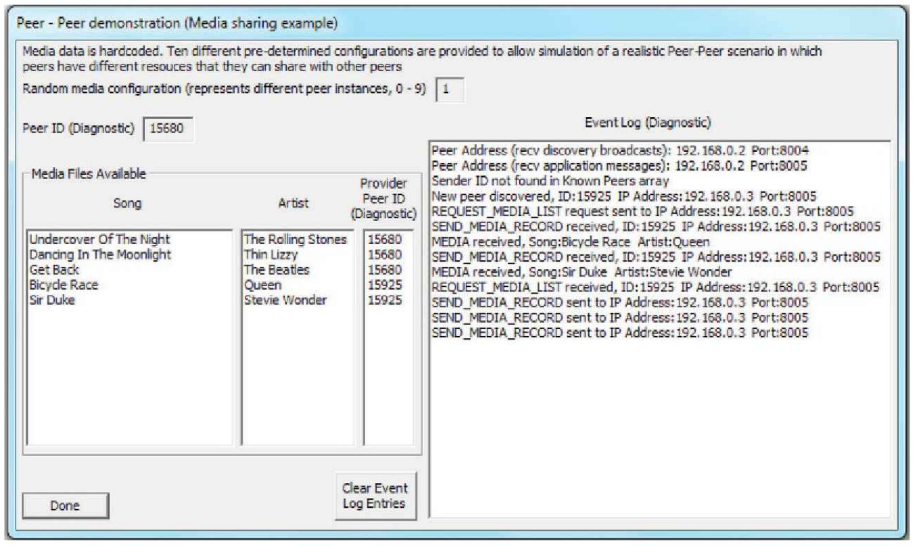
Discovery of a new peer automatically triggers sending a request message to that peer (REQUEST_MEDIA_LIST message type) causing the peer to respond with a series of data messages (SEND_MEDIA_RECORD message type), each one containing details of one media data item. Note that in the demonstration application, only the song title and artist name are transferred, but in a real media-sharing application, the actual music file would be able to be transferred on demand.

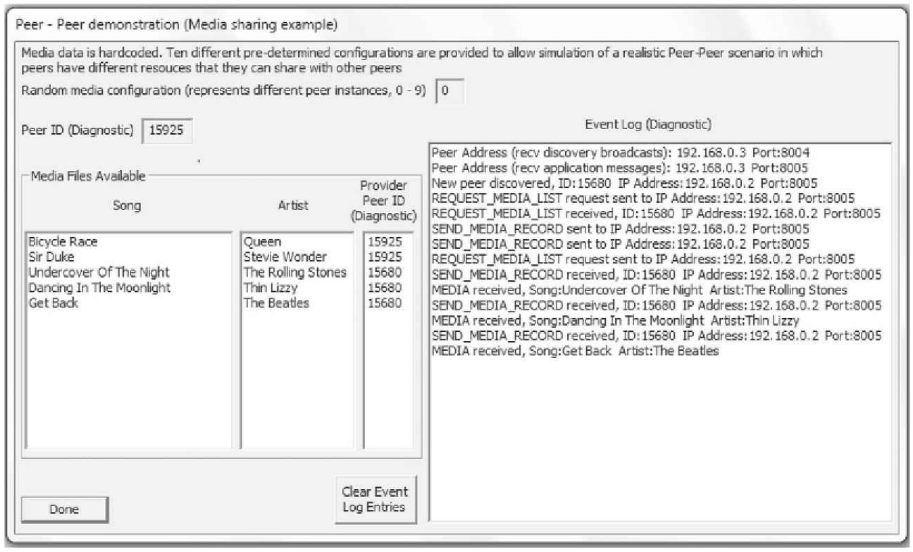
Expected Outcome for Part B

You should see the automatic peer discovery activity in progress, followed by the transfer of available media resources information between the peers. The diagnostic information would not be shown in a real application, since users do not need to know which peers provide which resources, or even be aware of the existence of the remote peers, or see any other information concerning the structure or connectivity of the application itself. Each user would only see the media resources available to them and would be able to access them in the same manner regardless of location, that is, the application provides access and location transparency.

The screenshots below show the two peer instances on two separate computers after they have each discovered the other peer. The first image is of the same peer as shown in part A above (it was left running) and the second image is of the second peer that was started on a different computer during part B of this activity.

ACTIVITY A2 EXPLORATION OF THE PEER-TO-PEER ARCHITECTURE—Cont'd





Method Part C: Understanding the Application-Level Protocol

This step involves further investigation to reinforce understanding. In particular, the focus is on the application-level protocol, that is, the sequence of messages passed between the components, the message types, and the message contents.

Part C1. Run the application several times with two peers each time. Look closely at the entries displayed in the activity log. From this, try to map out the application-level protocol (i.e., the message sequence) that is used in the mutual discovery and subsequent mutual exchange of media data between the peers.

ACTIVITY A2 EXPLORATION OF THE PEER-TO-PEER ARCHITECTURE—Cont'd

Hints: Each peer has the same program logic. The peer discovery and media exchange operate independently in each direction, due to the purposeful symmetrical design that was possible in this case and that makes the application simpler to design, develop, and test. Therefore, it is only necessary to map out the messages necessary for one peer (peer A) to discover another peer (peer B), to request peer B to send its media data, and for peer B to respond to that request by actually sending its data. The same sequence occurs when peer B discovers peer A.

Part C2. Run the application with at least three peers. Check that your mapping of the application-level protocol is still correct in these more complex scenarios.

In addition to empirical evaluation by running the application, you can also examine the application source code, which is provided, to confirm your findings and also to inspect the actual contents of the various message types.

A diagram showing the application-level protocol message sequence is provided in an Appendix at the end of the chapter so that you can check your findings.

Reflection

This activity has supported empirical exploration of two important aspects of peer-to-peer applications: firstly, a demonstration of automatic discovery between components, in which each peer keeps appropriate information in a table so that it can differentiate newly discovered peers from previously known peers and, secondly, an investigation of transparency requirements and transparency provision in distributed applications. The aspects of transparency that are most pertinent in this particular application are that the user does not need to know which other peers are present, where they are located, or the mapping of which music files are held at each peer. In a fully transparent application, the user will see only a list of available resources; if the diagnostic information were removed from the demonstration application's user interface, then this requirement would be satisfied.

These aspects of mechanism and behavior are of fundamental importance to the design of a wide variety of distributed applications. Repeat the experiments and observe the behavior a few more times if necessary until you understand clearly what is happening and how the behavior is achieved.

5.10 DISTRIBUTED OBJECTS

The distinguishing characteristic of the distributed objects approach is that it divides the functionality of an application into many small components (based on the objects in the code), thus allowing them to be distributed in very flexible ways across the available computers.

From the perspective of the number of components created, the distributed objects approach might show some resemblance to the multitier architectures discussed earlier. However, there are important differences in terms of the granularity and the functional basis on which the separation is performed. Object-oriented code separates program logic into functionally related and/or specific data-related sections (the objects). In most applications, components are deployed at a coarser grain than the object level. For example, a client (in a CS or multitier application) may comprise program code that is internally broken down into multiple objects at the code level, but these objects run together as a coherent single process at the client component level and similarly for the server component(s). In contrast, distributed objects target a fine-grained division of functionality when creating the software components, based on partitioning the actual code-level objects into separate components. Therefore, when comparing the distributed objects and multitier approaches, we should generally expect to see the distributed objects implementation comprises a larger number of smaller (or simpler) objects, perhaps each performing only a specific single function.

The component-location emphasis of the two approaches is also different. Multitier architectures are primarily concerned with the software structure and the division of the business logic over the various components, which is fundamentally a design-time concern. The distributed objects approach better supports run-time decisions for the placement of instances of components, as it operates on the level of individual objects and can take into account their specific resource requirements at run time.

However, a main challenge of such fine division of functionality is the number of connections between the components and the amount of communication that occurs. There may be opportunities to improve performance by ensuring that pairs of components that interact intensively are kept located on the same physical computer.

Figure 5.25 provides an illustrative distribution of seven objects across five computers, as part of a distributed banking application. The object location may be dynamic based on resource availability, and the communication relationships between components may change over time; hence, the caption describes the configuration as a snapshot (it may be different at other times). Notice that there can be multiple objects of the same type (such as "Customer account" in the example) as well as objects of different types spread across the system. The figure also shows how the services of one particular object (such as "Authentication" in the example) may be used by several other objects on demand. The objects communicate by making method calls to one another, for example, in order to execute a foreign currency transaction, the "Foreign currency transaction manager" component may need to call methods in the "Exchange rates converter" object.

Figure 5.25 also illustrates some of the differential benefits of the distributed objects architecture compared with multitier. In the example shown, the functionality of an application has been divided across many software components. The benefit of this is the flexibility with which the components can be distributed, for example, to be located based on proximity to specific resources. The location of

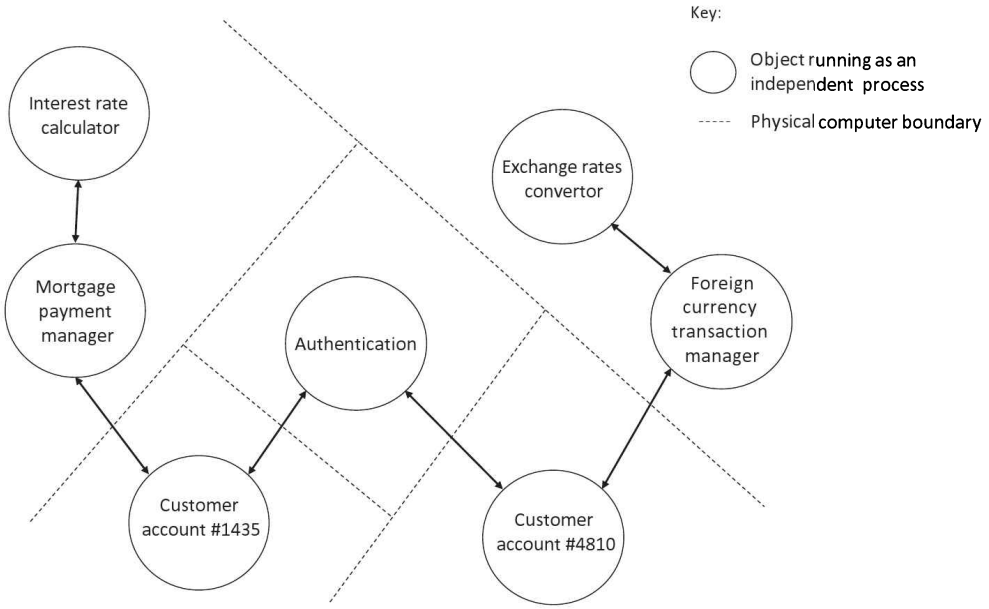


FIGURE 5.25

A distributed objects application; a run-time snapshot.

objects can also be performed on a load balancing basis. For example, if there are several objects (of the same type) in the system that each require a lot of processing resource, the distributed objects approach allows these objects to be executed at different physical sites (processors). Whereas, with the multitier architecture, if all these objects require the same type of processing, they would all have to queue to be processed by a particular server component that provides the requisite function.

Some infrastructure is necessary to support a distributed objects environment. Specific support requirements include a means for objects to be identified uniquely in the system, a means for objects to locate each other, and a means for objects to communicate (make remote method calls to each other). Middleware is commonly used to provide this support; see the next section.

5.11 MIDDLEWARE: SUPPORT FOR SOFTWARE ARCHITECTURES

This section deals with the way in which middleware supports software architectures. A more detailed discussion of the operation of middleware and the ways in which it provides transparency to applications is provided in Chapter 6.

Middleware is essentially a software layer that conceptually sits between processes and the network of computers. It provides to applications the abstraction of a single coherent processing platform, hiding details of the actual physical system including the number of processors and the distribution and location of specific resources.

Middleware provides a number of services to applications, to support component-based software architectures in which many software components are distributed within the system, and need assistance, for example, to locate each other and pass messages. Middleware is very important for dynamic software systems, such as distributed objects, because the location of the actual software components may not be fixed or at least not design time-decided.

5.11.1 MIDDLEWARE OPERATION, AN OVERVIEW

The following discussion of how middleware works focuses on the general principles of operation because there are actually many specific middleware technologies with various differences and special features for particular types of systems (examples include support for mobile computing applications and support for real-time processing).

Middleware provides a unique identifier for each application object that is supported in the system. When these objects are instantiated as running processes, the middleware keeps track of their physical location (i.e., what actual processor they are running on) in a database that itself may be distributed across the various computers in the system (as are the processes that constitute the middleware itself). The middleware may also keep track of the physical location of certain types of resource that the application uses.

Based on its stored information of which objects (running as processes) and resources are present, and where they are located, the middleware provides transparent connectivity services, essential to the operation of the applications in the system. A process can send a message to another process based only on the unique ID of the target process. The middleware uses the process ID to locate the process and deliver the message to it, and it then passes any reply message back to the sender process. This is achieved without either process having to know the actual physical location of the

other process, and it operates in the same manner whether the processes are local to each other (on the same processor) or remote. This transparent and dynamic location of objects (processes) also enables movement of objects within the system or for objects to be closed down in one location and subsequently to be run at another location; the middleware will always know the ID of the object and its current location.

It is important to recognize that the middleware plays the role of a communication facilitator. The middleware is not part of any of the applications it supports, but an external service. As such, the middleware does not participate in the business-level connectivity within the application; it does not understand the meaning or content of messages or indeed the specific roles of the various application objects present.

By using the middleware to facilitate connectivity, the objects do not need to have any built-in information concerning the address or location of the other components that they communicate with, and the applications do not need to support their own component discovery mechanisms (an example of such is found in the MediaShare_Peer example discussed earlier). Hence, by using the middleware services, applications can take advantage of run-time dynamic and automated connectivity. This is a form of loose coupling in which the middleware is the intermediary service. This has the benefits of flexibility (because components can be placed on processors based on dynamic resource availability) and robustness (because the application is not dependent on a rigid mapping of components to physical locations). A further important benefit is that the design of the application software is simplified by not having to manage connectivity directly; the value of this benefit increases dramatically for larger applications with many interacting objects.

Figure 5.26 depicts how the presence of the middleware as a virtual layer hides details of the true location of processes and makes them equally accessible regardless of actual location. This means that a process need not know where other processes that it communicates with are physically situated in the system. A process is decoupled from the underlying platform even if it is physically hosted on it. From a logical viewpoint, all processes are equally visible from any platform and all platform resources are equally accessible to all processes.

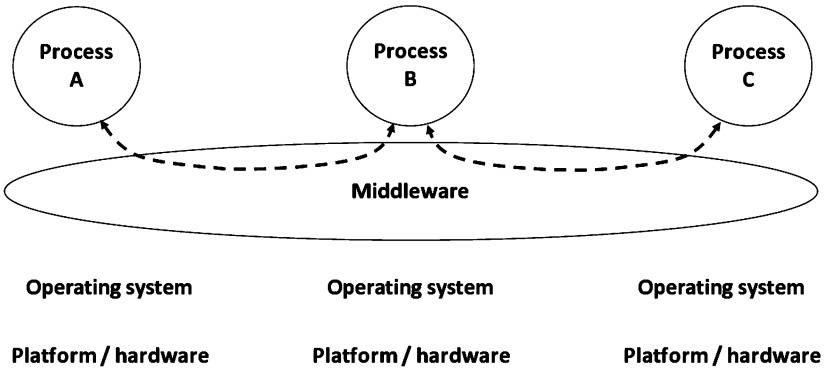


FIGURE 5.26

Overview of middleware.

5.12 SYSTEM MODELS OF COLLECTIVE RESOURCES AND COMPUTATION RESOURCE PROVISION

There are ongoing evolving trends in the ways that computing resources are provided. This section deals with the various models of computing provision that are important to the developers of distributed applications.

To implement successful distributed applications, in addition to the design of the software architectures of the applications themselves (which is the main concern of this book), there is also a need to consider carefully the design of the systems of computers upon which these applications run. The software developer will probably not be directly involved in the selection of processing hardware and the network technologies that link them together. However, it is nevertheless important for software developers to understand the nature of the various common system models and the impacts these may have on the level of resource available to applications, as well as issues such as robustness and communication latency that may have impacts on the overall run-time behavior.

To fully understand the various models of computation resource provision in the modern context, it is helpful to briefly consider some aspects of the history of computer systems.

Starting with the advent of the PC in the early 1980s, the cost of computing power was suddenly in the reach of most businesses where previously it had been prohibitively expensive. Initially, most applications were stand-alone and were used for the automation of mundane yet important tasks such as managing accounts and stock levels, as well as electronic office activities such as word processing. Local area networks became commonplace within organizations a few years later (towards the end of the 1980s), and this revolutionized the types of applications that were used. Now, it was possible to access resources such as databases and physical devices such as printers that were located at other computers within the system. This initial remote access to resources progressed to distributed computing in which the actual business logic of the applications was spread across multiple components, enabling better use of processing power throughout the system, better efficiency by performing the processing locally to the necessary data resources, and also the ability to share access to centrally held data among many users in scalable ways.

The next step was the widespread availability of connections to the Internet. This allowed high-speed data transfer between sites within organizations and between organizations themselves. Applications such as e-mail and e-commerce and online access to data and services revolutionized the role of the computer in the workplace (and elsewhere).

During this sequence of events, the actual number of computers owned by organizations was growing dramatically, to the point where we are today with almost every employee across a very wide range of job roles having a dedicated computer and relying on the use of that computer to carry out a significant proportion of their job tasks.

In addition to the front-end or access computers, there are also the service provision computers to consider. In the days of stand-alone computing, the resources were all in the one machine. In the days of remote access to resources, via local networks, the computers that hosted resources were essentially the same in terms of hardware configuration and levels of resource as the access computers. In fact, one office user could have remote access to a printer connected to a colleague's computer with exactly the same hardware specification as their own. However, once distributed computing became popular, the platforms that hosted the services needed to be more powerful: they needed more storage space

on the hard disks, more memory, faster processors, and faster network links. Organizations became increasingly dependent on these systems, such that they could not tolerate downtime, and thus, expert teams of systems engineers were employed. The service-hosting resources and their support infrastructure, including personnel, became a major cost center for large organizations, requiring specialist management.

In addition, most organizations involved in business, finance, retail, manufacturing, and service provision (such as hospitals and local government), in fact just about all organizations, have complex computer processing needs, requiring many different applications running and using a wide variety of different data resources. Prioritizing among these computing activities to ensure efficient usage of resources can be very complex, and thus, the simple resource provision model of just buying more and more expensive server hosting platform computers becomes inadequate, and a more structured resource base is needed.

As a result of these challenges, several categories of resource provision systems have evolved in which resources are pooled and managed collectively to provide a computing service. The various approaches differently emphasize a number of goals that include increased total computing power available to applications (especially the cluster systems); private application-specific logical grouping of resources to improve the management and efficiency of computing (especially the grid systems); the provision of computing as a service to reduce the cost of ownership of processing and storage (especially the data centers); and large-scale, robust, virtual environments where computing and storage are managed (especially the cloud systems).

5.12.1 CLUSTERS

Cluster computing is based on the use of a dedicated pool of processing units, typically owned by a single organization, and often reserved to run specific applications. The processors are usually loosely connected (i.e., connected by a network; see Section 5.4) and managed with special software so that they are used collectively and are seen by the clients of the cluster as a single system.

5.12.2 GRIDS

Grid computing is based on physically distributed computer resources used cooperatively to run one or more applications. The resources may be owned by several organizations and the main goal is the efficient processing of specific applications that need access to specific resources, for example, there may be data resources held at various locations that must all be accessed by a particular application. To ensure the performance and efficiency of the applications' execution, these resources can be brought together within a structure with common management and dedicated processors (a grid).

Typically, grids are differentiated from cluster computing in that the former tend to have geographically distributed resources, which are also often heterogeneous and are not limited to physical computing resources but may include application-specific resources such as files and databases, whereas the cluster resources are fundamentally the physical processors themselves and are more localized and likely to be homogeneous to provide a high-performance computing platform.

5.12.3 DATA CENTERS

Data centers are characterized by very large collections of processing and storage resources owned by a service provision company. Processing capacity is offered as a service to organizations that need large pools of computing resource. Typical data centers have thousands of processing units, so an organization can in effect rent as many as needed to run parallel or distributed applications.

The use of data centers reduces the cost of ownership of processing and storage resources because organizations can use what they need, when they need it, instead of having to own and manage their own systems. A particular advantage arises when an organization needs an additional large pool of resource for a short time, and the data center can rent it to them immediately without the time lag and cost of having to set up the resource in-house. The longer-term costs associated with leasing resources rather than owning them may be further compensated by the fact that the user organizations do not need to dedicate large areas of air-conditioned space to locally host services and also do not have to be concerned with expanding systems over time or continually upgrading hardware platforms and performing operating software updates. In addition, they also do not suffer the indirect costs such as workplace disruption and technical staff retraining associated with the hardware and software updates.

5.12.4 CLOUDS

Cloud computing can be thought of as a set of computing services, including processing and storage, which is provided in a virtualized way by a service provider. The actual processing resources are usually provided by an underlying data center, but the cloud concept provides transparency such that the infrastructure and its configuration are invisible to users.

In addition to the use of cloud facilities for processing, there is currently a lot of emphasis on the storage of bulk data, and this is increasingly popular for use with mobile devices and mobile computing applications. Collections of media files such as videos, songs, and images can be massive in terms of storage requirements and easily exceed the storage available on users' local devices such as tablets and smartphones. The cloud facilities let users upload their files into very large storage spaces (which are relatively huge compared with the capacities of their physical devices) and also offer the advantage of the users being able to access their files from anywhere and also to share them with other users.

The emphasis of cloud computing is towards an extension of personal computing resource (but shifted into a centralized, managed form) in which the cloud storage is allocated permanently to specific users. In contrast, the emphasis of the data center approach is more towards a rentable on-demand computing service. In some respects, a cloud system can be thought of as a set of services hosted on a data center infrastructure.

5.13 SOFTWARE LIBRARIES

Simple programs with limited functionality may be developed as a single source file containing the program code. Such programs are becoming increasingly rare as the extent of functionality and complexity of applications rises. Additional complexities arise from aspects that include multiple platform support, connectivity with other components, and user customization. There is also a growing trend

Figure 5.37 shows that the conventional process run-time environment (the process-to-operating system interface) is provided differently by each operating system (as explained in detail in Section 5.3 earlier). This means that applications must be built specifically for their target operating system, and therefore, executables are not transferrable across operating systems.

Figure 5.38 shows that the JVM provides a standard interface to application processes, regardless of the underlying system comprising any combination of operating system and hardware platform. This is achieved by using a special universal code format called the Java bytecode, which is a simple and standardized representation of the program logic with a low-level format that is similar to assembly language in some respects. The end result is that the user process in left part of Figure 5.38 will run correctly in the system shown in the right of Figure 5.38 without any modification or recompilation.

The JVM itself does have to be built specifically for the platform on which it runs, hence the differences in the JVM type, and the operating system interface to the JVM (the layer below the JVM), in Figure 5.38. The operating system sees the JVM as the schedulable entity; in other words, the JVM itself is the process that runs directly on top of the operating system and not the Java application.

5.15 STATIC AND DYNAMIC CONFIGURATIONS

The issue of static versus dynamic software configurations has been touched upon in several earlier sections, including Section 5.5.1 and also in Section 5.9 where ad hoc application configurations were discussed and explored in an activity. Later in this chapter, Section 5.16.1 also examines dynamic configuration of a service to mask failure of an individual server instance.

This section brings together the various issues associated with the choice between static and dynamic configurations of distributed applications. The main aspects that need to be considered are the way in which components locate each other, the way in which components form connections or at least communicate with each other, and the way in which roles are allocated to individual components in a group (e.g., when there are several instances of a service and a single coordinator is needed).

5.15.1 STATIC CONFIGURATION

Static configuration is achieved by default when building multicomponent applications. If a design fixes the roles of components and the ways in which the components relate to each other, connect to each other, and perform processing as part of the wider application, then you will end up with a static configuration. In such situations, there is no need for any additional services to dynamically map the components together; the one configuration is deemed to suit all situations. If you run the application a number of times, you will always arrive at the same component-to-component mapping.

The identity of software components can be represented in several different ways. Components need unique IDs that can be based, for example, on their IP address or on a URL, the URL being more flexible because it allows for relocation of services while still identifying the service components uniquely. Component IDs can also be system-wide unique identifiers allocated by services such as middleware, specifically for the purpose of mapping components together when messages need to be passed between them.

The essence of static configuration is that components connect to one another based on design time-decided mappings. This is most obvious if a direct reference to the identity of one component is built into another and used as the basis of forming connections or at least sending messages.

Statically configured applications are generally simpler to develop and test than their dynamically configured counterparts. However, they rely on complete design-time knowledge of their run-time behavior and also the environment in which they run. This is very difficult to confirm unless the application has very limited functional scope and comprises a small number of components. This also means that statically configured systems may need more frequent version updates to deal with any changes in the run-time environment, because each specific configuration is fixed for a particular setting. Perhaps, the most significant limitation of static configuration is that it is inflexible with respect to dynamic events such as component failures or failure of the specifically addressed platform where a resource is located.

5.15.2 DYNAMIC CONFIGURATION

Dynamic configuration of distributed applications increases run-time flexibility and potentially improves efficiency and robustness. In the case of efficiency, this is because components can be moved between physical locations to better map onto resource availability, or component requests can be dynamically diverted to different instances of server processes (e.g., to balance the load on a group of processors). Robustness is improved because component-to-component mappings can be adjusted to account for events such as the failure of a specific component; so, for example, if a client is mapped to a particular instance of a service and that instance fails, then the client can be remapped to another instance of the same service. There are mechanisms that can be used to do this automatically.

The essence of dynamic configuration is that components discover one another based on roles rather than component-unique IDs. That is, components know what types of services they require and request connections to components that provide those services without knowing the IDs of those components in advance. Typically, additional services are required to facilitate this, specifically to advertise the services provided by components, or otherwise to find services based on a role description, and to facilitate connectivity between the two processes. Examples of external services that facilitate dynamic discovery and connectivity include name services and middleware.

Some services perform dynamic configuration internally, for example, to balance load or to overcome or mask failures of individual subcomponents of the service. Mechanisms such as election algorithms are often used to automatically select a coordinator of a dynamic group of processes whose membership can change over time, as different nodes join and leave the service cluster. An election algorithm follows several steps: firstly to detect that a particular component has failed, secondly to carry out an election or negotiation among the remaining components to choose a new coordinator, and finally to inform all components of the identity of the new coordinator to suppress additional unnecessary elections. Election algorithms are discussed in detail in Chapter 6. Other mechanisms for dynamic configuration discussed elsewhere in the current chapter include the use of heartbeat messages by which one component informs others of its ongoing presence and health status and service advertisement messages that facilitate automatic discovery of components or the services they offer.

Dynamic configuration mechanisms also facilitate context-aware behavior in systems (see next section).

5.15.3 CONTEXT AWARENESS

Context awareness implies that an application's behavior or configuration takes the operating context into account, that is, it dynamically adjusts its behavior or configuration to suit environmental or

operating conditions. This is a complex aspect of systems behavior, which is out of scope for the book generally, but is very briefly introduced here for completeness.

Context is information that enables a system to provide a specific, rather than generic response. To give a very simple example, you ask me the weather forecast for tomorrow and I respond "it will rain." This is only of use to you if we have previously communicated the location we are referring to (which contextualizes both the question and the answer in this case). One example relating to distributed services concerns the failure of a server instance. The number of remaining servers is a very important context information because if there are still one hundred servers operating, the failure of one is relatively insignificant, but if there are only one, or none remaining, then it is serious and new servers need to be instantiated.

Dynamic configuration mechanisms and context information provide a powerful combination to enable sophisticated and automated reconfiguration responses to events such as sudden load increases or failure of components.

5.16 NONFUNCTIONAL REQUIREMENTS OF DISTRIBUTED APPLICATIONS

There are a number of common nonfunctional requirements of distributed applications, most of which have already been mentioned several times in various contexts in the book. Here, they are related and differentiated.

Robustness. This is a fundamental requirement of almost all distributed applications, although it can be interpreted in different ways. Robustness in the sense that there are no failures at the component level is unrealistic because no matter how well designed your software is, there are always external factors that can interrupt the operation of a distributed application. Examples include excessive network traffic causing delay or a time-out, a network failure isolating a server from its clients, and a power failure of the computer hosting the server. Therefore, a key approach to robustness is to build in redundancy, that is, to have multiple instances of critical components such that there is no single point of failure (i.e., there is no single component that, if it fails, causes the system itself to fail).

Availability. This requirement is concerned with the proportion of time that the application or service is available for use. Some business- or finance-related services can cost their owners very large sums of money for each hour that they are unavailable; consider stock-trading systems, for example. Some systems such as remote monitoring of dangerous environments such as power stations and factory production systems are safety critical, and thus, availability needs to be as near to 100% as possible. You may come across the term "five nines" availability, which means that the goal is for the system to be available 99.999% of the time. Availability and robustness are sometimes confused, but technically, they are different; an example is scheduled maintenance time in which a system is not available, but has not failed. Another example is where a system can support a certain number of users (say, 100). The 101th user connects to the system and is denied service, so the service is not available to him specifically, but it has not failed.

Consistency. This is the most important of all of the nonfunctional requirements. If the data in the system do not remain consistent, the system cannot be trusted and so has failed at the highest level. To put this into context with a very simple example, consider a banking application in which customers can transfer money between several of their own accounts online (over the Internet). Suppose a customer has £200 in her current account and £300 in her savings account. She moves £100 from the

current account into the savings account. This requires multiple separate updates within the bank's system, possibly involving two different databases and several software components. Suppose the first step (to remove £100 from the current account) succeeds, but the second step (to place it in the savings account) fails; the system has become temporarily inconsistent because the customer's credit has gone from £500 to £400 when the total should have remained at £500. If the system has been designed well, it should automatically detect that this inconsistency has arisen and "roll back" the state of the system to the previous consistent state (in this case the initial balance values).

Performance (or Responsiveness). This is the requirement that a transaction is handled by the system within a certain time frame. For user-driven queries, the reply should be timely in the context of the use of the information. For example, if the application is stock trading, then a response within a second or so is perhaps acceptable, whereas if it is an e-commerce system in which one company is ordering wholesale batches of products from another, a longer delay of several seconds is adequate as the transactions are not so time-critical. See also the discussion on Scheduling for Real-Time systems in Chapter 2.

Consistent performance is also important for user confidence in the system. Variation in response times impacts some aspects of usability, since long delays can frustrate users or lead to input errors (e.g., where a user is uncertain that the system has detected a keystroke and so enters it again, this can lead to duplicate orders, duplicate payments, and so forth).

Scalability. This is the requirement that it should be possible to increase the scale of a system without changing the design. The increase could be in terms of the number of concurrent users supported or throughput (the number of queries or transactions handled per unit of time). Scaling up a system to meet the new demand could involve adding additional instances of components (see Section 5.16.1), placing performance-critical components on more powerful hardware platforms, or redesigning specific bottleneck components in isolation, but should not require a redesign of the overall system architecture or operation.

It is important not to confuse scalability and responsiveness; a lack of scalability as load increases may be the underlying cause of reduced responsiveness, but they are different concerns.

Extensibility. It should be possible to extend the functionality of a system without needing to redesign the system itself and without impact on the other nonfunctional requirements, the only permissible exception being that there may be an acceptable trade-off between the addition of new features and a corresponding reduction in scalability as the new features may lead to an increase in communication intensity.

Transparency. This is the requirement that the internal complexities of systems are hidden from users such that they are presented with a simple to use, consistent, and coherent system. This is often cited as the fundamental quality metric for distributed systems.

A general transparency goal is the single-system abstraction such that a user, or a running process that needs access to resources, is presented with a well-designed interface to the system that hides the distribution itself. All resources should appear to be available locally and should be accessed with the same actions regardless if they are truly local or remote. As mentioned in numerous other sections, there are many different flavors of transparency and a wide variety of ways to facilitate it.

Transparency provision is potentially impacted by the design of every component and the way they interact. It should be a main theme of concern during the requirements analysis phase because it is not generally possible to post-fit transparency mechanisms to an inherently nontransparent design.

Usability. This is a general label for a broad set of concerns that overlap several of the other nonfunctional requirements, especially responsiveness and transparency. Usability is also related to some specific technical aspects, such as the quality of user interfaces and consistency in the way information is presented, especially if there are several different user interfaces provided by different components.

In general, the more transparency that is provided, the more usable a system is. This is because users are shielded from having to know technical details of the system in order to use the system. Usability in turn improves overall correctness because with clear, easy to use systems, in which users do not have to follow complex procedures and are not asked to make decisions based on unclear situations, there are fewer mistakes made by users or technical managers.

5.16.1 REPLICATION

Replication is a commonly used technique in distributed systems, not only having the potential to contribute to several of the nonfunctional requirements identified above, most notably robustness, availability, and responsiveness, but also having the potential to disrupt consistency if not implemented appropriately.

This section focuses on the relationships between replication and the nonfunctional requirements of distributed systems. Transparency and mechanistic aspects of replication are discussed further in Chapter 6.

The simplest form of replication is service provision replication where there are multiple server instances, but each supports its own set of clients, so there is no data sharing across the servers. This approach is used to improve availability and responsiveness but does not improve robustness as there is still only one copy of the state of each client session and only one copy of the data used by each client. There is no requirement for data update propagation, but instead, there is a need to direct the clients to servers so as to balance the load across them, since if all clients connect to one server, then the service provision replication has no effect on performance.

Consider the use-case game. Multiple copies of the server could be started at different locations to achieve a simple form of replication without the need for any modification to the current design. This replicates function, but not data or state. Each client would still connect to a specific server instance (identified by its IP address) and thus would only see the other players connected to the same server instance, advertised in the available players list. More users in total could be supported, that is, availability is enhanced, and because users could connect to the geographically nearest server, the network latency may be reduced, increasing responsiveness. However, the clients are associated, via their server, in isolated groups. There is no mechanism to move games between servers or to back up game state at other servers; each client is still dependent on a particular server to hold the game state, so robustness is not improved in regard to active games.

Where replication of data occurs, the extent of performance benefits and the extent of the challenge to manage data and ensure data consistency are related to the data access mode supported, specifically whether data can be modified or is read-only, during transactions. If the replication is of active application state, or of updateable resources, then the additional complexity can be very high due to the need to propagate updates to all copies. This is relatively simpler if only one copy of the shared data is writable by the user application, but even so, there is still a significant challenge of ensuring that copies that go off-line are brought up-to-date correctly when they reappear. There are three common scenarios that occur:

1. The data are read-only at all of the replicas. This could apply to information-provision applications such as an online train timetable system, in which multiple servers could be used to support a great many clients retrieving train journey details simultaneously. However, the data itself can only be modified by a separate system (a management portal not available to the public users). There are no user queries that can cause any data updates. In such cases where the replication is across multiple server instances of read-only resources, then the additional complexity arising from the replication itself is relatively low and the scheme can scale very well.
2. The replication is implemented such that only a single copy of the data is writable and the other copies are readable. The challenges here are ensuring that only one copy of the data really is ever writable at any specific time and also ensuring that all updates to the writable copy are copied promptly and reliably to the other copies. This can work well in applications in which read accesses significantly outnumber write accesses.
3. The replication is implemented such that all copies of the data are writable. This can lead to various issues, including the specific problem of a lost update, where two copies of a particular data value are both modified in parallel. When each of the changes is then propagated across the system, the one that is applied last will overwrite the one that was applied first, that is, the first update has been lost. The lost update problem is significant because it occurs even when all components are functioning correctly, that is, it is not caused by a failure, but rather is an artifact of the system usage and the specific timing of events.

This approach can be very complex to manage and is generally undesirable because in addition to the lost update problem, there a number of ways the system can become inconsistent.

The primary objective when adding replication is to ensure that the system remains consistent under all circumstances and use scenarios. This must always be the case, to ensure the correctness of systems even when the original motivation and purpose for implementing replication are to improve availability, responsiveness, or robustness.[3]

Figure 5.39 illustrates a simple form of replication in which a single instance of the service (and thus data) is available to users at any given time. This is because all service requests are directed to the master instance of the service, so the backup instance is effectively invisible to users. When a user request causes data held by the server to change, the master instance updates its local copy of the data and also propagates any updates to the backup instance. The backup instance monitors the presence of the master instance. This could, for example, be facilitated by the master instance sending a status massage to the backup instance periodically, in which case if these messages cease, the backup instance becomes aware that the master instance has failed. Figure 5.39 shows the situation when both instances of the service are healthy and updates are performed on both replicas of the data. The mechanics of step 1 (parts a and b) in Figure 5.39 could be based on multicast or directed broadcast or by using a group communication mechanism, which is discussed in Chapter 6.

[3]Replication mechanisms provide a very good example of the general rule that whenever a new feature is added to an application or system, for whatever reason, it potentially introduces additional challenges or new failure modes. Circumventing the new challenges or protecting against these new failure modes can be costly in terms of design and development effort and especially testing (which must cover a wide state space of possible scenarios). Ultimately, we can find ourselves doing what the author terms "complexity tail chasing" in which we add increasing layers of complexity to deal with the issues arising from the complexity of earlier layers. Therefore, before adding any additional mechanisms such as replication, it is vital that a detailed requirements analysis has been carried out and that the consequences of adding the mechanism are considered and a balanced decision is made through in-depth understanding of the effective costs and benefits.

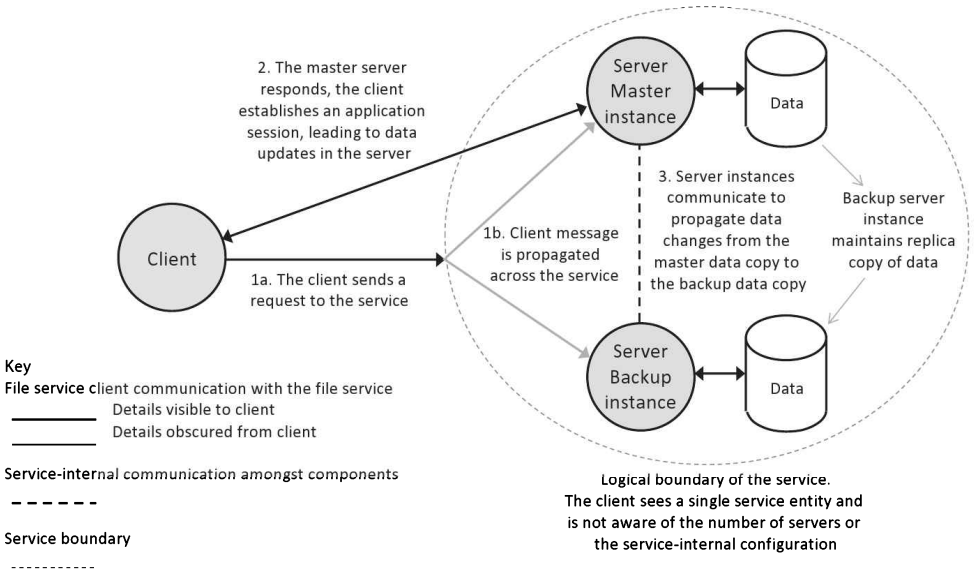


FIGURE 5.39

Master-backup data replication of active application state and data.

Figure 5.40 illustrates the adaptation of the configuration of the replicated service that occurs when the master server instance has failed. Initially, there will be a period in which the service is unavailable, because the backup instance has not yet detected the previous master's failure, so has not taken over the master role. Once the backup server instance detects that the master has failed, it takes over as master (this is also described as being promoted to master state). From this point onward, service requests are dealt with by this server instance.

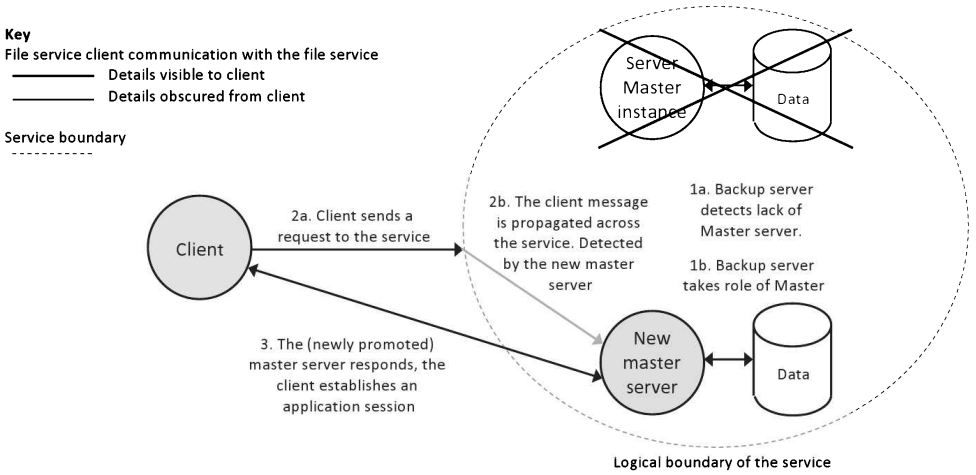


FIGURE 5.40

Backup-promotion scenario in master-backup data replication.

The mechanism by which the failure of the master instance is detected must itself be reliable. The detection is usually based on some form of heartbeat mechanism such that the healthy master server sends a periodic massage to inform the backup that it is still healthy. This may be omitted when updates are occurring regularly (because the update messages also serve the purpose of informing the backup that the master is still healthy, without adding any additional communication) and only switched on if the interval between updates exceeds some threshold. Thus, the backup copy should not wait more than a specified period without a message from the master. However, messages can be lost or delayed by the network itself, and thus, it is not safe to trigger the backup copy switching over on the omission of a single heartbeat message; perhaps, three omitted messages in sequence are appropriate in some systems. The configuration of the heartbeat scheme (which includes the interval between messages and the number of messages that must be missed in sequence to serve as confirmation that the master has failed) must be tuned for the specific application as it represents a three-way trade-off between the responsiveness of the backup system, the message overheads to maintain the backup system, and also the risk of a false alarm, which introduces a further risk of data inconsistency if both copies become writable simultaneously.

5.16.2 SEMANTICS OF REPLICATION

When implementing mechanisms such as replication, it is important to consider carefully the merits of the various alternative ways of achieving the required functionality. For replication, semantics are concerned with the actual way the replication mechanisms behave in terms of the way they manipulate the system data. Issues that need to be considered include the following:

- What should happen when the master copy becomes unavailable, should the backup copy become available for access, or is it maintained only to establish the system consistency once the master server has been repaired?
- If the backup copy is made available (upon master failure), there is the issue as to whether it should be made read-only or read-write; the main concern is that the two copies could get out of sync. This could happen if updates occur at the backup copy, and then, it also fails; when the previous master copy comes back on line, it has no knowledge of the intervening transactions.
- Are the roles of master and backup preassigned or do they arise dynamically from the system behavior (e.g., the most accessed instance may be automatically assigned to be the master instance)? If the roles were preassigned, then what should happen after a master that had failed (and hence the initial backup instance is now acting as master) recovers? Does the original master reclaim its master role, or does it assume backup status?
- Some implementations of replication employ an odd number of server instances (at least three) so that in the event that the copies become out of sync, then a vote can be taken, that is, the correct value is taken to be the majority value. However, there are no absolute guarantees that the majority are correct (it is less likely, but possible that two out of the three missed a particular update). There is also no guarantee that there will even be a majority subset (consider a group of five server instances, in which one has failed and the four remaining are split with two having a particular data value and the other two having a different value).

Figure 5.40 illustrates one of the possible fallback scenarios when the master copy fails. In the approach illustrated, upon detection of master failure, the backup instance promotes itself to master

status (i.e., it takes over the role of master). Data consistency is preserved so long as the previous master instance had propagated all data updates to the (then) backup copy, which it should do if it is operating correctly.

5.16.3 AN IMPLEMENTATION OF REPLICATION

A demonstration application is provided to illustrate the implementation and operation of replication in the context of a simple database application. The same database theme as used in Activity A1 is continued, but in this case, there is a single database that is replicated, instead of two different databases, as used in the earlier activity. In this specific example, the replication has been used to ensure the service is robust against server-process failure. A single master status instance of the service deals with client service requests. The master instance also broadcasts periodic heartbeat messages to signal its presence to other instances of the service. An additional instance can exist, having the status of backup; its role is to monitor the status of the master instance by receiving heartbeat messages. If the absence of three successive heartbeat messages is detected, then the backup instance promotes itself to master status. The master instance of the service propagates any data updates made by the client, to the backup instance, therefore keeping the data held at the backup instance synchronized with the master instance and thus keeping the service consistent. If the backup copy has to take over from the master instance then, the data will be up-to-date and reflect any changes made before the master instance crashed.

In order to show the variety of design choices available, the implementation uses an alternative means of clients locating the service to that shown in Figures 5.39 and 5.40. The technique shown in those examples is based on the client sending a service discovery broadcast message and the master instance of the service responding. Instead, the implementation of the demonstration replicated service uses service advertising in which the master instance of the service sends broadcast messages at regular short intervals of a few seconds. A newly started client must wait to receive a service advertisement message that contains the IP address and port details needed to connect to the service. If the backup server instance elevates itself to master status, it takes over sending the service advertisement broadcasts, so that clients always detect the current master instance of the service. In this implementation, a failed master instance that recovers will take on the status of backup (assuming that another instance now has master status, such as the previous backup instance).

Figure 5.41 shows the service-internal behavior that determines the state of each instance of the replicated database application. The diagram shows the behavior of a single process; each process maintains its own copy of the state transition logic and represents its state internally as a pair of variables. In this case, the state variables that govern the state transition behavior are as follows:

m_ServerState {Backup, Master}	An enumerated type
m_iHeartbeatMissedCount ≥ 0	An integer

Each time a heartbeat is detected within the expected time frame, the m_iHeartbeatMissedCount variable is reset to 0. When a time-out occurs (the heartbeat was not detected in the expected time frame), the variable is incremented. If the variable reaches the value 3, the backup instance elevates itself to master status. There are two ways in which a master status instance can cease to exist; firstly, if it crashes or is purposely removed from the system and, secondly, if it detects any heartbeat messages from another master instance (in which case it demotes itself to backup status). This is a fail-safe mechanism to prevent the coexistence of multiple master instances.

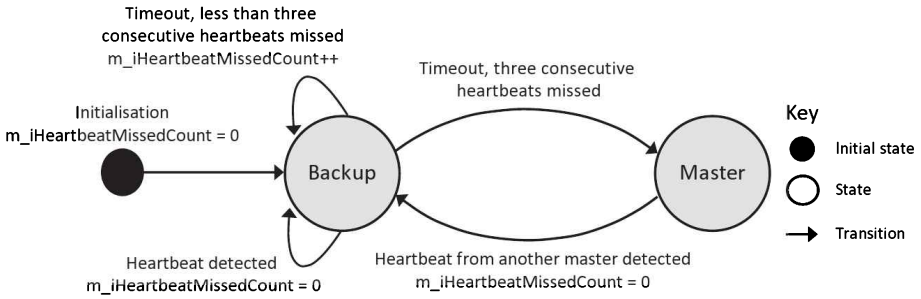


FIGURE 5.41

The state transition diagram of the replicated database demonstration application.

There are two types of communication within the replication demonstration application: communication within the service itself (between the server instances) and communication between the application client and the master instance of the service. Figure 5.42 shows the socket-level connectivity between the various components.

Figure 5.42 shows the socket-level communication within the replication demonstration application. The communication between the application client and the master instance of the service is shown to the left. It comprises the server sending service advertisements, which are broadcast over UDP and used by the client to gain the server's address details and thus connect using TCP. The TCP connection is used for all application requests and replies between the client and the server. The communication between the master instance of the service and the backup instance is shown to the right. Both processes have the same sockets, but the broadcast sockets and the TCP sockets used to connect to clients are inactive on whichever instance is in backup state. Only the master instance broadcasts heartbeat messages, which the backup instance receives. When the client has caused an update of the data held by the master instance, the master sends an update propagation message to the backup instance.

The behavior of the replication demonstration application is explored in Activity A3. The full source code is available as part of the book's accompanying resources.

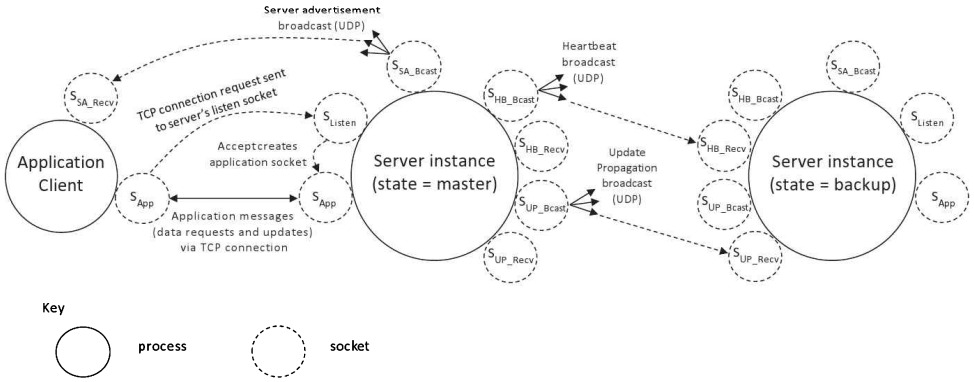


FIGURE 5.42

The socket-level connectivity between the components of the replication service.

ACTIVITY A3 EXPLORING THE MASTER-BACKUP DATA REPLICATION MECHANISM

A replicated database application is used to investigate the behavior of a data replication mechanism, as well as aspects of dynamic service configuration and component role allocation through the use of heartbeat messages, data update propagation between server instances, and service advertisement broadcasts to enable clients to locate the master service instance.

This activity uses the Replication_Client and Replication_Server programs. To simplify the configuration of the software and to ensure that the focus of the experimentation is on the architectural aspects of replication, the database servers actually hold their data in the form of in-memory data tables. The values are initially hard-coded, but can be updated through client requests. Any updates that occur at the master server instance are then propagated to the backup server instance.

During the experiments, observe the diagnostic event information displayed in each component, which provides details of internal behavior.

Prerequisites

Two networked computers are needed because each instance of the service needs to bind to the same set of ports; therefore, they cannot be coresident on the same computer. Since broadcast communication is used, the computers need to be in the same local network.

Learning Outcomes

- To understand the concept of data replication
- To become familiar with a specific replication mechanism
- To explore the behavior of a simple replicated database application
- To explore automatic service configuration and component role allocation
- To explore dynamic service discovery with service advertisement broadcasts
- To explore update propagation
- To gain an appreciation of failure transparency in a practical application setting

The activity is performed in five stages.

Method Part A: Component Self-Configuration and Self-Advertisement (One Server)

This part of the activity is concerned with the self-configuration that occurs when a single server instance of the replicated database service is started.

Start a single copy of the client program Replication_Client and the server program Replication_Server on different computers. The server process initializes itself to backup state. It listens for heartbeat messages, and because it does not detect any (it counts three consecutively missed heartbeats), it elevates itself to master state. Once in master state, it begins to broadcast periodic service advertisements. This is to enable any clients to locate the server.

Initially, the client process does not know the address of the server, so it cannot send a connection request. Notice how it receives a service advertisement message (containing the server's address details), updates its record of the server's address, and then uses this information to automatically connect to the service.

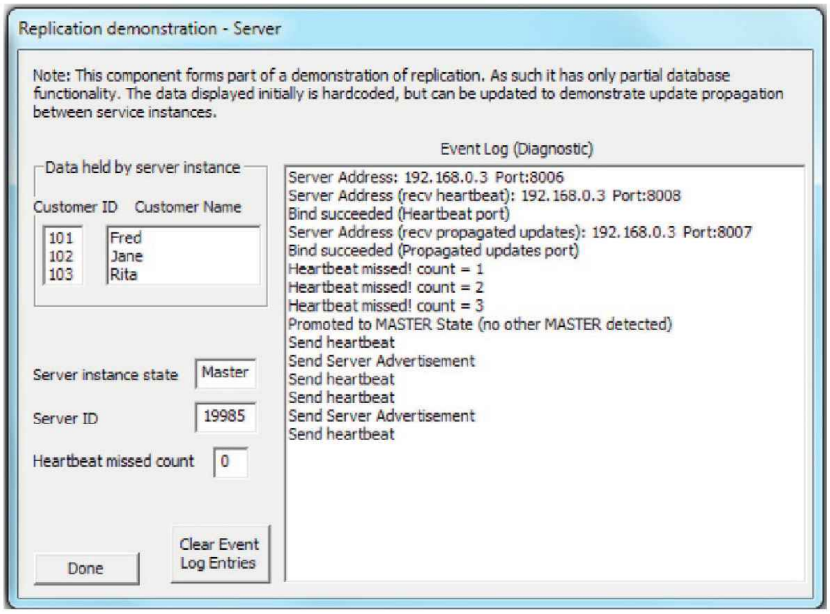
Expected Outcome for Part A

You should see the server initializes itself to backup state, then waits for heartbeat messages and on receiving none, and elevates to master status. At this point, you see it starts to send service advertisement and heartbeat messages.

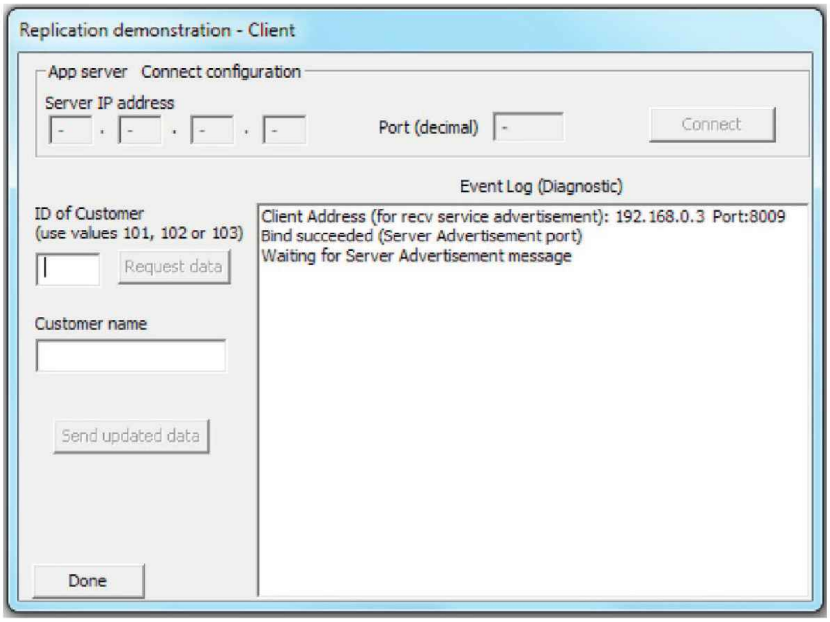
The client process receives the server advertisement.

The screenshot below shows the server instance initializing to backup state and then elevating to master state, at which point it begins sending heartbeat messages and server advertisement messages.

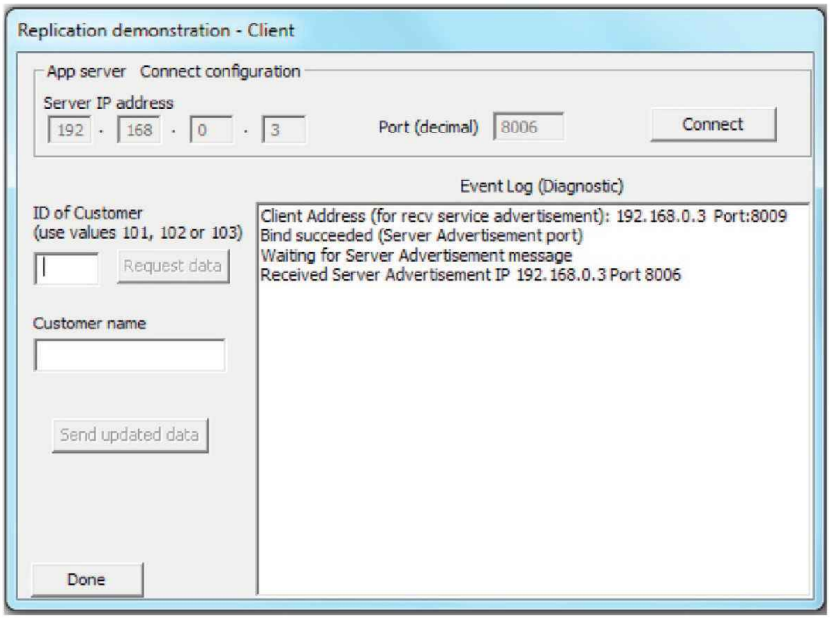
ACTIVITY A3 EXPLORING THE MASTER-BACKUP DATA REPLICATION MECHANISM—Cont'd



The following screenshots show the client component waiting for the service advertisement message and then receiving it and updating the server address details it holds.



ACTIVITY A3 EXPLORING THE MASTER-BACKUP DATA REPLICATION MECHANISM—Cont'd



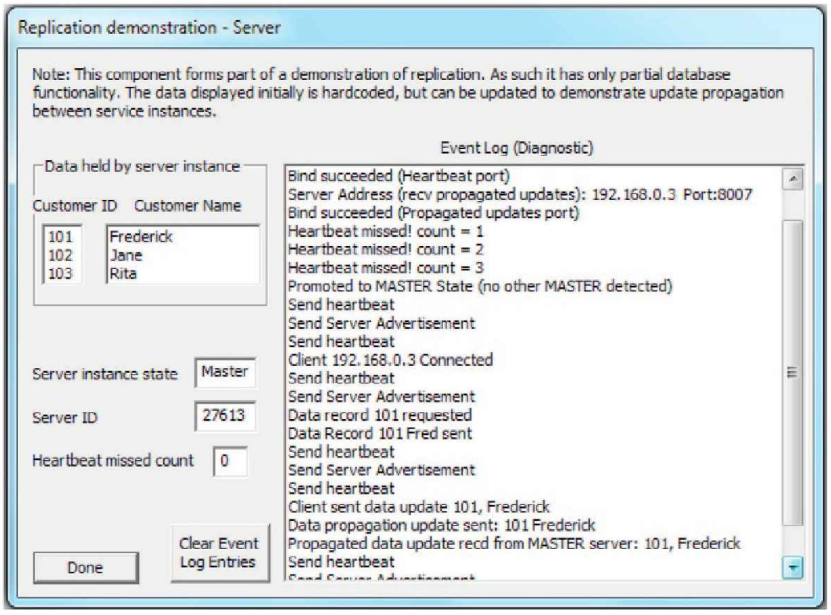
Method Part B: Using the Database Service

The application can now be used. Experiment with connecting the client to the server and requesting data items from the server. Data values held at the server can also be updated; try a few requests and a few updates.

Expected Outcome for Part B

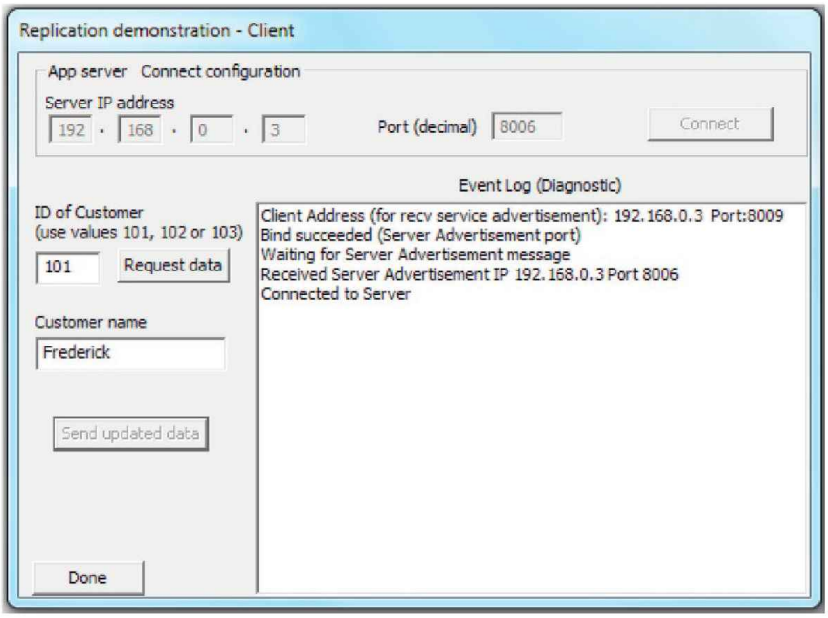
The client should connect to the server (it establishes a TCP connection). You should be able to perform data requests and updates at the client and see the effect of these at the server.

The screenshot below shows the server state after the client has connected, and the customer number 101 name has been changed by the client from "Fred" to "Frederick."



ACTIVITY A3 EXPLORING THE MASTER-BACKUP DATA REPLICATION MECHANISM—Cont'd

The screenshot below shows the corresponding state of the client interface after the data value was updated and sent to the server.



Method Part C: Component Self-Configuration (Two Servers)

This stage explores the self-configuration that occurs when there are two server instances present.

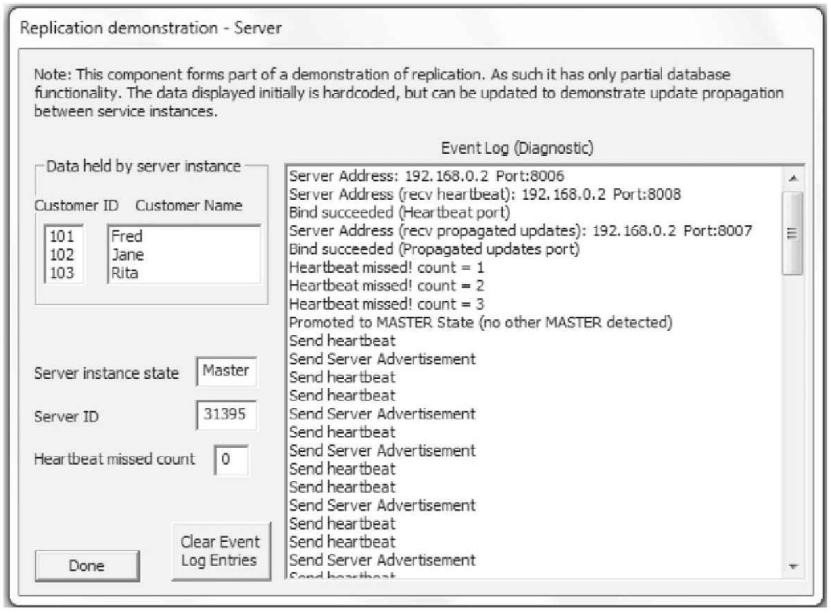
Start two instances of the service on different computers. One process should promote itself to master status (the same event sequence as explored in part A); the second instance then detects the heartbeat messages from the master process, causing it to remain in the backup state.

Expected Outcome for Part C

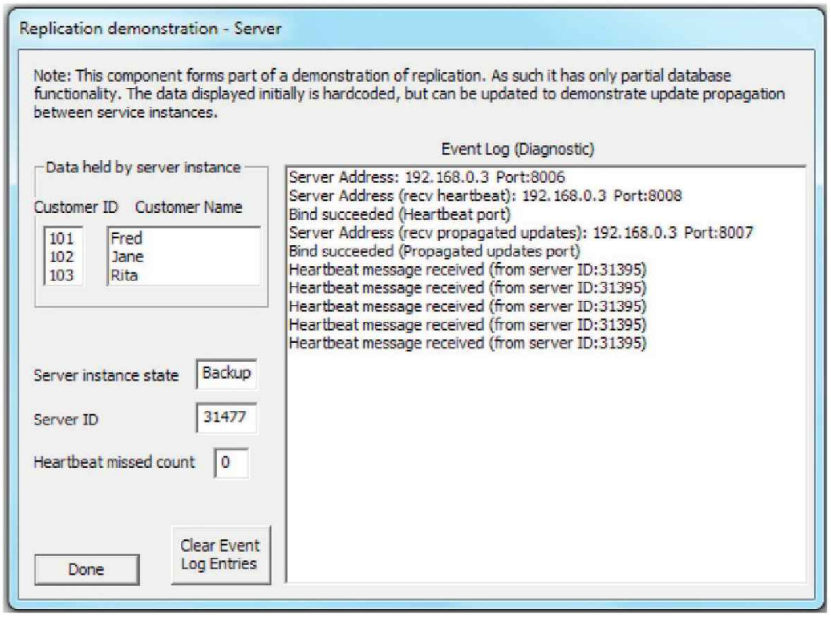
You should see that, by using the heartbeat mechanism, the service autoconfigures itself so that there is one master instance and one backup instance.

This screenshot shows the service instance that elevates to master. Typically, it is the one that is started first, because each instance waits the same time period to detect heartbeat messages, and the first one to start waiting is the first one to get to the end of the waiting period and elevate.

ACTIVITY A3 EXPLORING THE MASTER-BACKUP DATA REPLICATION MECHANISM—Cont'd



This screenshot (below) shows the server instance that remains in backup state because it is receiving the heartbeat messages from the master instance.



ACTIVITY A3 EXPLORING THE MASTER-BACKUP DATA REPLICATION MECHANISM—Cont'd

Method Part D: Update Propagation in Operation

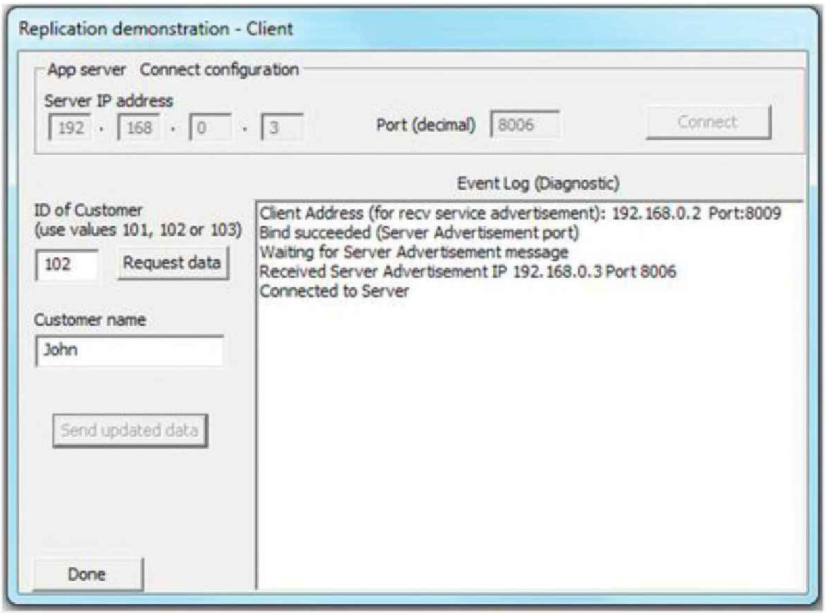
Start two instances of the service on different computers (or continue on from part C).

Start a client on the same computer as one of the server instances, or use a third computer. The client will receive service advertisement messages from the master instance, and therefore, always connect to master; confirm this experimentally.

Notice that the customer with ID 102 initially has the name "Jane." Use the client to change this to "John" (request the data, then manually edit the data in the client user interface, and then use the "Send updated data" button to send the new value back to the master server instance).

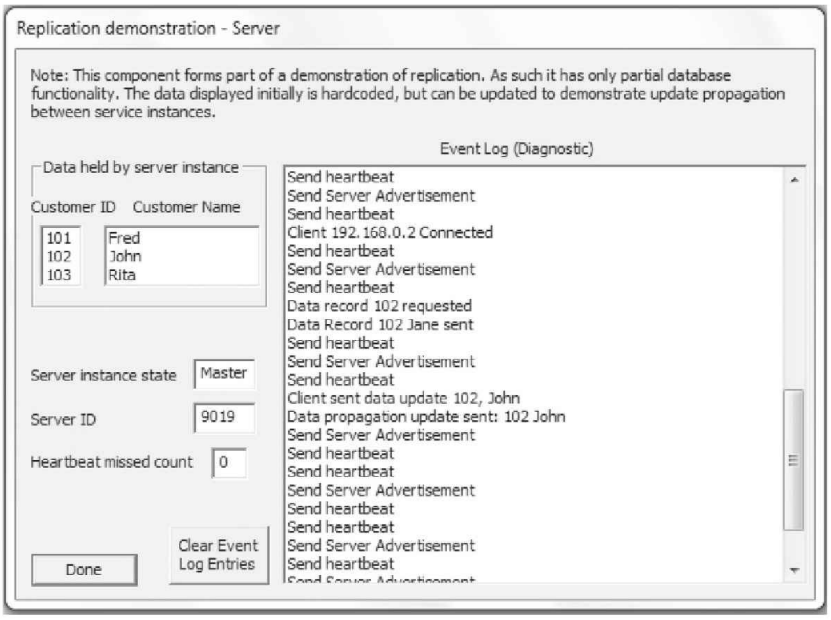
Expected Outcome for Part D

You should see that the update is performed successfully by the client, using the request-change-send sequence described above. The client interface is shown in the following screenshot.

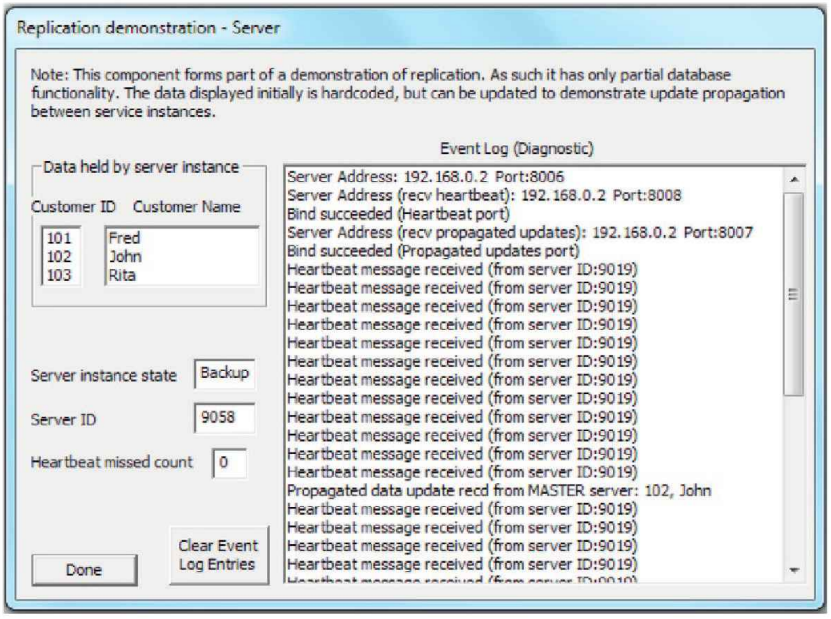


The master server instance receives the update request message from the client, performs the update on its own copy of the data, and also propagates the update to the backup server instance (as shown in the screenshot below) to maintain data consistency.

ACTIVITY A3 EXPLORING THE MASTER-BACKUP DATA REPLICATION MECHANISM—Cont'd



The backup server instance receives the propagated update from the master instance and updates its copy of the data accordingly; see the screenshot below.



ACTIVITY A3 EXPLORING THE MASTER-BACKUP DATA REPLICATION MECHANISM—Cont'd

Method Part E: Automatic Reconfiguration and Failure Transparency

This stage explores the behavior when the master instance fails, and the backup server detects the lack of heartbeat messages.

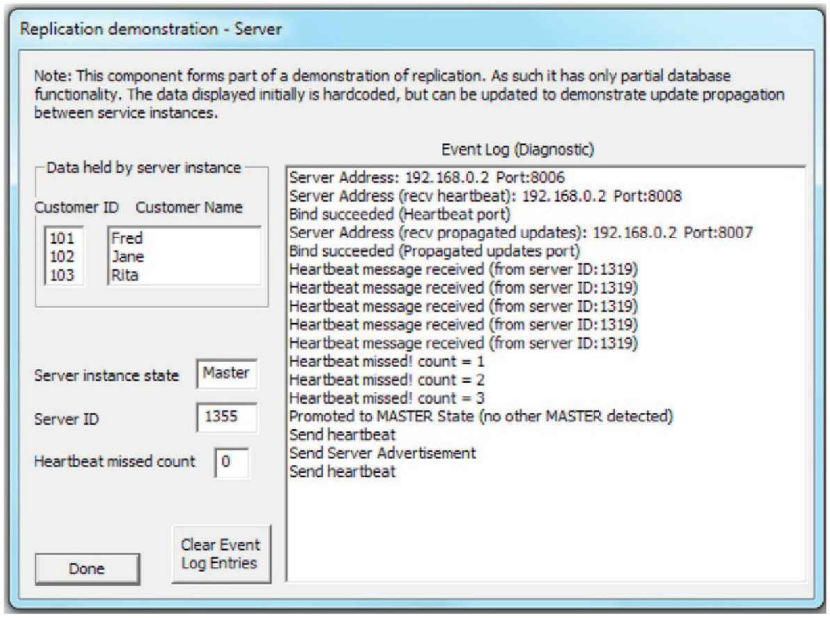
Start one instance of the service on each of two different computers. Watch while the service self-organizes (i.e., one of the servers becomes the master instance, and the other remains in backup state).

Now, close the master instance (simulating a crash). Observe the behavior of the backup instance. It should detect that the master is no longer present (there are no heartbeat messages) and thus promote itself to master status.

Once the new master has established itself, restart the other server (which was originally the master). Upon starting up, it should detect the heartbeat messages being sent by the current master instance, and thus, the restarted server should remain in the backup state.

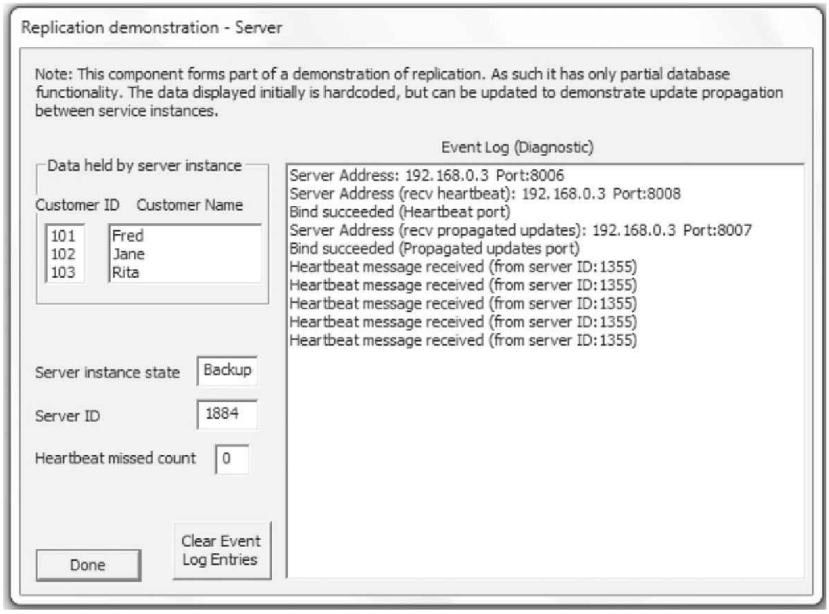
Expected Outcome for Part E

The screenshot below shows the backup service instance detecting the absence of the master: when three consecutive heartbeats have been missed, it promotes itself to master status and begins to broadcast heartbeat messages and server advertisements.



When the other instance of the service (which was previously the master and then crashed) is restarted, it detects the heartbeat messages from the current master and thus assumes backup status, as shown in the screenshot below. Note the new server ID of the restarted instance (1884), instead of its original ID (1319) when it was running previously (evidenced in the screenshot above when it was sending heartbeat messages). This is because the ID is randomly generated when the server process starts. Note also the ID of the other (now master) instance remains unchanged (1355).

ACTIVITY A3 EXPLORING THE MASTER-BACKUP DATA REPLICATION MECHANISM—Cont'd



Reflection

This activity facilitates exploration of data replication in a realistic application setting. Through practical experimentation, you can see how several important mechanisms operate (service advertisement, heartbeat messages, and data update propagation). These are used to achieve automatic component discovery and service self-configuration, as well as the higher-level achievement of active data replication to achieve a robust, failure transparent database service.

It is recommended that you experiment further with the replication demonstration application to gain an in-depth understanding of its various behaviors. You should also inspect the source code and try to marry up the behavior witnessed to the underlying program logic.

5.17 THE RELATIONSHIP BETWEEN DISTRIBUTED APPLICATIONS AND NETWORKS

Distributed systems add management, structure, and most significantly transparency on top of the functionality provided by the underlying networks to achieve their connectivity and communication. Therefore, distributed applications should be considered as being in a layer conceptually above the top of the network protocol stack and not as part of the application layer.

To put this into context, consider the File Transfer Protocol (FTP). This commonly used application-layer protocol can be realized in an application form, that is, a file transfer utility, and also can be embedded into applications that may integrate a file transfer function within their general operation (such as system configuration and management utilities and automated software installers). See Figure 5.43.

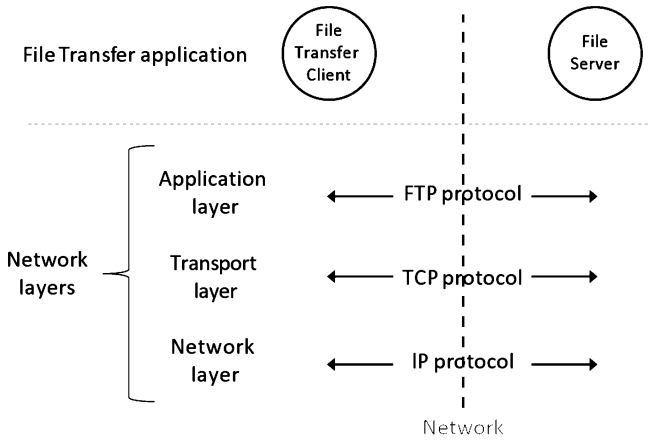


FIGURE 5.43

Distributed applications sit above the application layer.

Figure 5.43 shows the distinction between FTP, which is part of the network protocol stack, and a file transfer application. A file transfer application wraps the FTP functionality within an application form; it is software built on top of the FTP protocol and possibly integrates the functionality of transferring files with other functionality such as security and a bespoke user interface suitable for the intended use scenario. In the case of a file transfer utility, the client component is fundamentally a user interface enabling the user to submit commands to the server. The business logic is contained within the server component. The user must identify the file server by address when connecting, so this application provides limited transparency, and as such, it could be described as a network application rather than a distributed application.

Distributed applications use the connectivity provided by the network protocols as building blocks to achieve more sophisticated communication scenarios, especially by incorporating services and mechanisms such as name services and replication to achieve higher usability and especially transparency such that the user need not be aware of the distributed nature of the applications. For example, a file service that automatically mirrors changes in your local file system to a backup service in the cloud would have greater functionality than a file transfer utility and significantly would be far more transparent. See Figure 5.44.

Figure 5.44 illustrates a distributed file service application. The extent of transparency provided enables the client to make service requests without being aware of the configuration of the service itself. The client does not need to know the actual number of server entities, their location, or the way they communicate within the service, achieve a logical structure, and maintain data consistency. So long as the client's service requests are met, then all is well from the client's point of view.

Network printing provides a further example for discussion. The print-client component can be embedded into, or invoked by, various applications from which the user may need to print a document (such as word processors and web browsers). The print server is a process that runs on the remote computer, which is connected to the printer, or the server may be embedded into the printer itself if it is a stand-alone network printer with its own network identity. The client performs the role of sending documents to the printer. This is essentially a file transfer, but with added control information, which includes the portion of the document to print, the print quality to use, and the number of copies

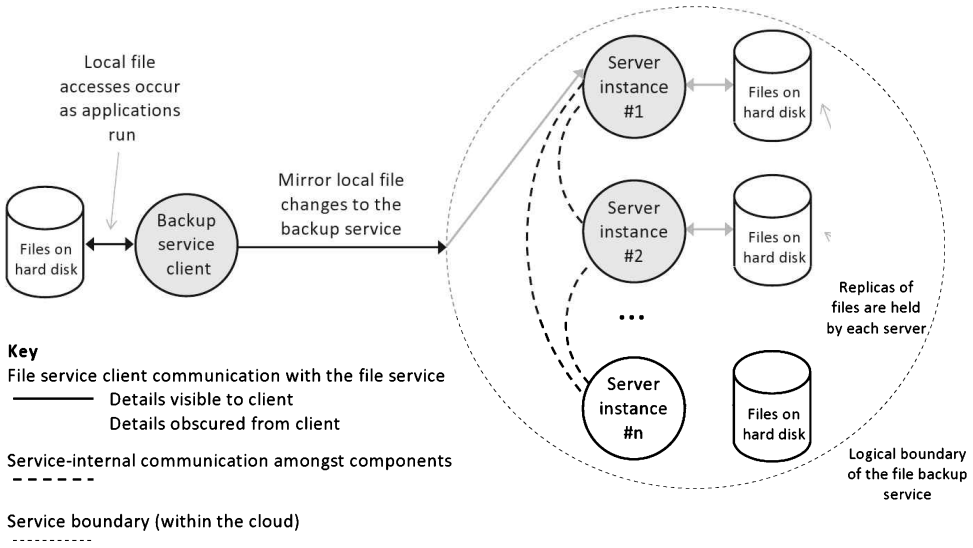


FIGURE 5.44

A file service as an example of a distributed application.

required. The main business logic (such as checking access rights, queue management, and logging print jobs) and the actual printing are managed within the print server component. As with a file transfer utility, a network printing facility has limited transparency and thus could be described as a network application rather than a distributed application.

5.18 TRANSPARENCY FROM THE ARCHITECTURE VIEWPOINT

The forms of transparency that are particularly important from the architecture viewpoint are briefly identified here and are treaded in more depth in Chapter 6.

Access transparency. The mechanisms to access resources should not be affected or modified by the software architecture of the application or by the type of coupling between components. The use of services to facilitate connectivity, such as name services and middleware, should not negatively impact access transparency.

Location transparency. Software architectures should be designed to support location transparency, especially where dynamic configuration and/or loose coupling is used. Location transparency should be maintained when components are relocated or when component roles change dynamically due to internal service reconfiguration.

Replication transparency. Replication of components is a key architectural technique to enhance robustness, availability, and responsiveness. This must be implemented such that the internal configuration and behavior within the replicated services (including details of server cardinality, server failures, and update propagation) are hidden from the external components that communicate with the services, such that they are presented with a single-instance view of the service.

Failure transparency. Failure transparency at the component level should be supported within the architectural design, using techniques such as loose coupling, dynamic configuration, and replication.

Scaling transparency. Software architectures should avoid unnecessarily complex or intense coupling between components. Tight coupling should be replaced with loose coupling using intermediate connectivity services where possible. The extent of interdependency and rigid structure between components impacts on maintainability and flexibility aspects of scalability.

Performance transparency. The intensity of communication between components should be considered carefully in the architectural design, because it can impact severely on performance. In general, the number of other components that each component communicates with should be kept to a minimum.

5.19 THE CASE STUDY FROM THE ARCHITECTURAL PERSPECTIVE

The game has a CS architecture. The game server has limited functional scope since it is intended primarily for demonstration and exploration; it only provides the business logic for a simple game, as well as the management of a list of up to ten connected users and a list of up to five simultaneously active games. The CS design is well suited to the game in its current form, simplicity being its main advantage in this case. There are only two-component types; the client provides the user interface and the server provides the business logic. At the small scale of this application, there are no issues with performance.

If the application were to be extended to support features such as user registration and authentication or multiple different games or if it needed to be scaled up to support large numbers of users, then a three-tier design may be more suitable because of its flexibility and better transparency provision.

5.19.1 STATEFUL SERVER DESIGN

The game is based on a stateful server design; the server stores the game state including which players are associated by a particular game and which of the players has the next turn to make a move. The server sends messages to each client containing details of the moves made by the opponent so that both clients display an up-to-date representation of the game-board positions. When designing the game application, it was necessary to weigh up the relative advantages and disadvantages of the stateful and stateless approaches, in the specific context of the game. The stateful server approach is the more natural choice for this particular application because the server provides a connection between pairs of clients playing a game and needs access to the game state to mediate between the clients in terms of controlling the game action; if the server did not store the state locally, then the messages passed through the server between the clients would need to additionally contain the state information. The negative aspect of stateful design in this case is that if the server were to crash, the game will be destroyed.

The stateless server alternative requires spreading the state across the two clients such that they each hold details of their own and their opponent's move positions. This approach would increase the complexity of both the client and the server components. The client becomes more complex as it has to manage the state locally, and the server becomes more complex because it is much harder to govern the game, in terms of preventing clients from making multiple moves at one turn, not taking a turn, or reversing previous moves. This is because the server would only see snapshots of the game from step to step, instead of following it/enforcing it on an alternating move-by-move basis, which it controls (as

in the case of the stateful design). In addition, the stateless server approach causes there to be separate copies of the state at each client, and therefore, it is possible for inconsistency to arise through the two sets of state becoming out of sync, such as if both clients think it is their turn to move at the same time (in which case the game is effectively destroyed anyway). The additional design and testing that would be required to ensure the game state remains consistent in all scenarios that could occur would far outweigh the single negative aspect of the stateful design in this case, and bear in mind that the stateless server approach is still sensitive to the failure of either client.[4]

5.19.2 SEPARATION OF CONCERNS FOR THE GAME COMPONENTS

The game comprises two-component types with clearly defined roles. The server keeps the application state organized in a two-level hierarchy. The lower level holds information concerning the connected clients (representing players), which is held within an array of connection structures (one structure per client). The higher level relates players to games in play using an array of game structures (one structure per game). The game structure also holds the game-grid state, which is updated as the server receives gameplay move messages from each client and forwards them to the client's opponent. A logical game is created when one player selects another from the "available players" list, which is updated and transmitted to each connected client when a new client joins, leaves, or becomes tied up in a game. The demonstration application supports up to five simultaneous games, by holding an array of five game structures. The connection and game structures have been presented in Chapter 3.

5.19.2.1 CS Variant

The game is an example of the fat-server variant of the CS architecture, in which the client provides the user interface logic, while the server provides the game logic as well as the data and state management. In this case, the processing demands on the server, per game (and thus per client), are quite low. The message arrival frequency is low, with game move messages arriving at typical intervals of at least two seconds or so, per game.

5.19.2.2 Client and Server Lifetimes

As with most CS applications, the game has been designed such that the client is the active component that connects to the server on demand of the user (i.e., when the user wishes to play a game, they will run the client process). To support on-demand connectivity, the server is required to run continuously, whereas the client has a short lifetime (the duration of a game session). The individual client lifetimes are independent of those of other clients, but of course, a client can only be linked in a game with another client that is present at the time of game initiation. To ensure meaningful game management, and to avoid wasting system resources, one of the roles of the server is to detect the loss of connection with one of the client processes and to close the game.

[4]For this particular application (in which games are played between pairs of clients), a peer-to-peer design could also be considered and would have similar merits to the stateless server design, except that there would be no server, so there would need to be a mechanism to facilitate clients discovering and associating with one another to establish games. The peer-to-peer design would encounter the same challenges of duplication of game state and the need to ensure consistency as the stateless server approach does.

5.19.3 PHYSICAL AND LOGICAL ARCHITECTURES OF THE GAME APPLICATION

Physical architecture is concerned with the relative distribution of components, the specific location of each and the way in which they are connected and communicate. In contrast, logical architecture is concerned with the identity of components and the communication mappings between components without knowledge of actual physical system configuration. This difference gives rise to two different architectural views of the game.

Figure 5.45 shows the physical view of the game application when a game between two users is in play. The two users (players) each have a computer on which they run the game client program (which causes an instance of the game client process to run on each computer). The game server program is running on a third computer (the running instance of this program is the server process). The computers are connected via the physical network and communicate using the IP protocol at the network layer. The processes communicate the game-application messages at the transport layer and thus could use either the TCP or the UDP protocols, each of which is encapsulated and carried across the network by the IP protocol. TCP was chosen for the game due to its reliability. The processes are identified for the purposes of the TCP communication by a combination of their host computer's IP address and the port number they are using, because this is the way in which the sockets (the communication end points) are differentiated. Figure 5.46 provides a logical view of this situation.

Figure 5.46 shows the logical view of the physical system that was shown in Figure 5.45. This can be mapped onto the transport layer of the network model, in which the communicating processes are identified by the ports they are mapped to. The separation between the processes is indicated by the dotted lines; this indicates that the processes are independent entities in the system. This could, but does not have to, designate the presence of the physical network. In other words, the logical view is only concerned with which processes communicate with which other processes and is not concerned with the actual locations of processes. The communication used in the game has been configured such that

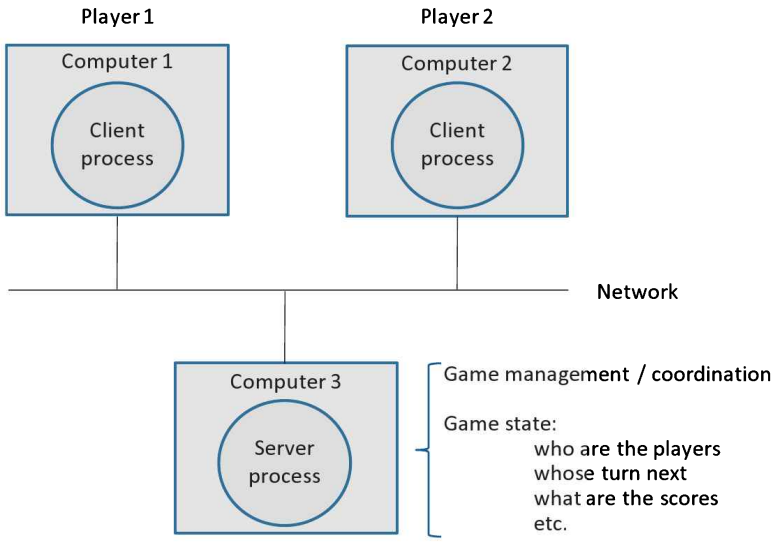


FIGURE 5.45

The physical architecture of the game.

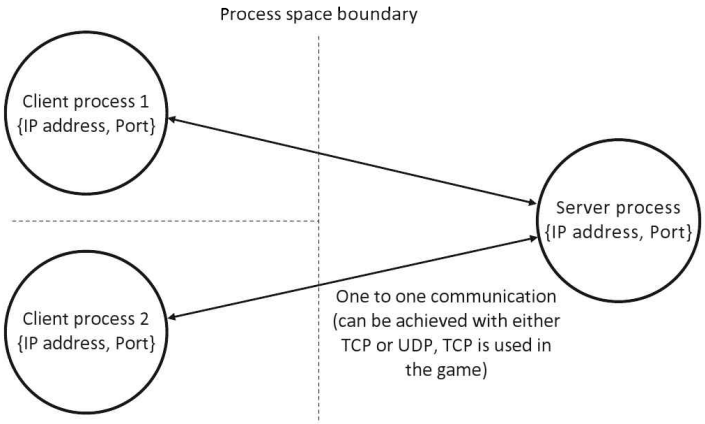


FIGURE 5.46

Logical view of the game connectivity.

processes can be located on the same or different computers because only the server process binds to the port that is used. At the transport layer, the network can be abstracted as a logical concept (because it facilitates process-to-process connectivity without concern for details of how it is achieved, such as which technologies are used or how many routers there are in the path). Both the TCP and the UDP transport layer protocols will deliver messages from process to process in an access transparent manner (this means that the mechanism to perform the send and receive actions is identical regardless of whether the two processes are on the same computer or different computers). See also the discussion of logical and physical views of systems in Chapter 3.

Server replication can be used to increase the robustness of an application such as the game, and it can also improve scalability and availability in large systems. However, replication also increases complexity, especially if state has to be propagated between the server instances. The example game does not need to scale up to large numbers of players and the service provided is not considered sufficiently important to justify the additional design, build, and test expense of adding server replication. However, if the game were to be part of an online casino, with paying customers expecting a highly robust and available service, then the decision would be different.

5.19.4 TRANSPARENCY ASPECTS OF THE GAME

Ideally, users of distributed applications are provided with an abstraction, which hides the presence of the network and the boundaries between physical computers such that the user is presented with a single-machine view of the system. Obviously, in a multiplayer game, the user is aware that there is at least one other user who is located at a different computer and that they are connected by a network; this is an example where full network transparency is not necessary or achievable. However, the user does not need to know any of the details concerning the connectivity, distribution, or internal architecture of the game application.

Locating the server: In the case study game, the user manually enters the IP address of the server. This is an issue from a transparency point of view because it reveals the distributed nature of the game and also requires the user to know, or find out, the address of the server. Techniques to automatically

locate the server, such as the server advertisement used in the replication demonstration earlier in the chapter, could be used. Adding server advertisement to the use-case game is one of the programming challenges at the end of Chapter 3.

6.1 RATIONALE AND OVERVIEW

This chapter takes a systems-level approach to the three main focal areas: transparency, common services, and middleware.

Distributed systems can comprise many different interacting components and as a result are dynamic and complex in many ways relating to both their structure and behavior. The potentially very high complexity of systems is problematic for developers of distributed applications and also for users and is a major risk to correctness and quality. From the developer's perspective, complexity and dynamic behavior make systems less predictable and understandable and therefore make application design, development, and testing more difficult and increase the probability that untested scenarios exist that potentially hide latent faults. From the user's perspective, systems that are unreliable and difficult to use or require the user to know technical details of system configuration are of low usability and may not be trusted.

Transparency provision is a main influence on the systems' quality and is therefore one of the main focal themes of this chapter. The causes of complexity and the need for transparency to shield application developers and users from it have been discussed in the earlier chapters in their specific contexts. The approach in this chapter is to focus on transparency itself and to deal with each of its forms in detail, with examples of technologies and mechanisms to facilitate transparency.

In keeping with the systems-level theme of this chapter, a second main focal area is common services in distributed systems. There are many benefits that arise from the provision of a number of common services, which are used by applications and other services. These benefits include standardization of the main aspects of behavior and reduction of the complexity of applications by removing the need to embed into them the commonly required functionalities that these services provide; this would be inefficient and not always possible. This chapter explores a number of common services in-depth.

The third main focus is on middleware technologies that bind the components of the systems together and facilitate interoperability. Middleware is explored in detail, as well as a number of platform-independent and implementation-agnostic technologies for data and message formatting and transport across heterogeneous systems.

6.2 TRANSPARENCY

It is no accident that transparency has featured strongly in all of the four core chapters of the book. As has been demonstrated in those chapters, distributed systems can be complex in many different ways.

To understand the nature and importance of transparency, consider two different roles that humans play when interacting with distributed systems. As a designer or developer of distributed systems and applications, it is of course necessary that the various internal mechanisms of the systems are understood. There are technical challenges that relate to the interconnection and collaboration of many components, with issues such as locating the components, managing communication between the components, and ensuring that specific timing or sequencing requirements are met. There may be a need to replicate some services or data resources while ensuring that the system remains consistent. It may be necessary to allow the system to dynamically configure itself, automatically forming new connections between components to adjust in order to meet greater service demand or to overcome the failure of a specific component. A user of the system has a completely different viewpoint. They wish to use the system to perform their work without having to understand the details of how the system is configured or how it works.

A good analogy is provided by our relationship with cars. Certainly, there are some drivers who understand very well what the various mechanical parts of the car are and how these parts such as the engine, gearbox, steering, brakes, and suspension work. However, the majority of drivers do not understand, and do not want to spend effort to learn, how the various parts work. Their view of the car is as a means of transport, not as a machine. During use, events can happen quickly. If someone steps into the path of the car, the user must press the brake pedal immediately; there is no time to think about it, and understanding the mechanical details of how the brake works is of no help in actually stopping the car in an emergency. The user implicitly trusts that the braking system has been designed to be suitable for purpose, usually without question. When the user presses the brake pedal, they expect the car to stop.

The designer has a quite different view of the car braking system. The designer may have spent a lot of effort ensuring that the brakes are as safe as possible. They may have built-in redundant components, such as the use of dual-brake pipelines and dual hydraulic cylinders to ensure that even if one component should fail, the car will still stop. In the designer's view, the braking system is a work of art, but one that must remain hidden. The user's quality measure, i.e., the measure of success, is that the car continues to stop on demand, throughout its entire lifetime.

The car analogy helps illustrate why transparency is considered to be a very important indicator of quality and usability for distributed systems. Users have a high-level functional view of systems, i.e, they have expectations of the functionality that the system will provide for them, without being troubled by having to know how that functionality will be achieved. If a user wishes to edit a file, then they will expect to be able to retrieve the file and display its contents on their screen without needing to know where the file was stored. When the user makes a change to the file and saves it, the file system may have to update several replica copies of the file stored at different locations: The user is not interested in this level of detail; they just want some form of acknowledgment that the file was saved.

A phrase often used to describe the requirement of transparency is that there should be a "single-system abstraction presented to the user." This is quite a good, memorable way of summing up all the various more specific needs of transparency. The single-system abstraction implies that users (and the application processes that run on their behalf) are presented with a well-designed interface to the system that hides the distribution of the underlying processes and resources. Resource names should be human-friendly and should be represented in ways that are independent of their physical location, which may change dynamically in some systems; of course, the user should not be aware that such reconfigurations are occurring. When a user requests a resource and service, the system should locate that resource and pass the user's request to the service, respectively, without the user having to know where the resource and service are located, whether it is replicated, and so forth. Problems that occur during use of the system should be hidden from the user (to the greatest extent possible). If one instance of a service crashes, then the user's request should be routed to another instance. The bottom line is that the user should get a correct response to their service request without being aware of the failure that occurred; it should not matter which actual instance dealt with the request.

The reason why transparency is so prominently discussed throughout this book is that it must be a first-class concern when designing distributed systems if the resulting system is to be of high quality and high usability. Transparency requirements must be taken into account during the design of each component and service. Transparency is a crosscutting concern and should be considered as an integral theme and certainly not as something that can be bolted on later. A flawed design that does not support transparency cannot in general be patched up at a later stage without significant reworking, which may lead to inconsistencies.

As transparency is such an important and far-reaching topic and with a great many facets, it is commonly divided into a number of transparency forms, which facilitates greater focus and depth of inquiry of the key subtopics.

6.2.1 ACCESS TRANSPARENCY

In a distributed system, it is likely that the users will routinely use a mix of resources and services, some of which are locally held at their own computer and some of which are held on or provided by other computers within the system. Access transparency requires that objects (this includes resources and services) are accessed with the same operations regardless of whether they are local or remote. In other words, the user's interface to access a particular object should be consistent for that object no matter where it is actually stored in the system.

A popular way to achieve access transparency is to install a software layer between applications and the operating system that can deal with access resolutions, sending local requests to the local resource provision service and remote requests to the corresponding layer at some other computers where the required object is located. This is described as resource virtualization because the software layer makes all resources appear to be local to the user.

Figure 6.1 shows the concept of resource virtualization. The process makes a request to a service instead of the resource directly. The service is responsible for locating the actual resource and passing the access request to it and subsequently returning the result back to the process. Using this approach, the process accesses all supported objects via the same call interface, whether they are local or remote, regardless of any underlying implementation differences at the lower level where the resources are stored.

An example of resource virtualization is Unix's virtual file system (VFS) that transparently differentiates between accesses to local files that are handled by the Unix file system and accesses to remote files that are handled by the Network File System (NFS).

Figure 6.2 shows how the VFS layer provides access transparency. When a request for a file is received, the VFS layer finds where the file is and directs requests for files that are held locally to the local file storage system. Requests for files that are held on other computers are passed to the NFS client program that makes a request for the file to the NFS server on the appropriate remote computer. The file

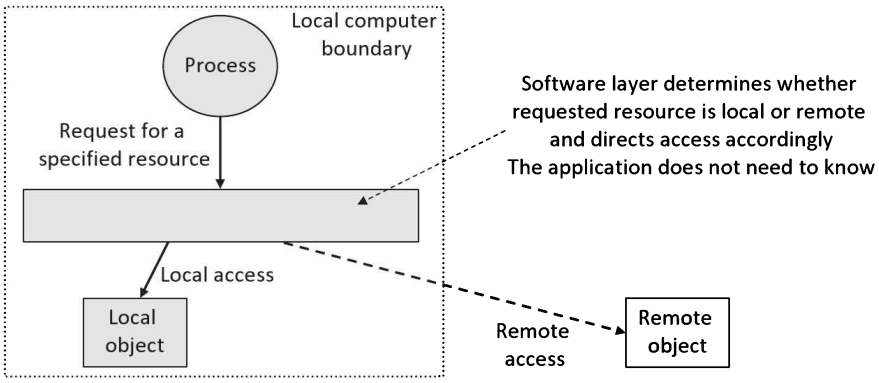


FIGURE 6.1

Generic representation of resource virtualization via a software layer or service.

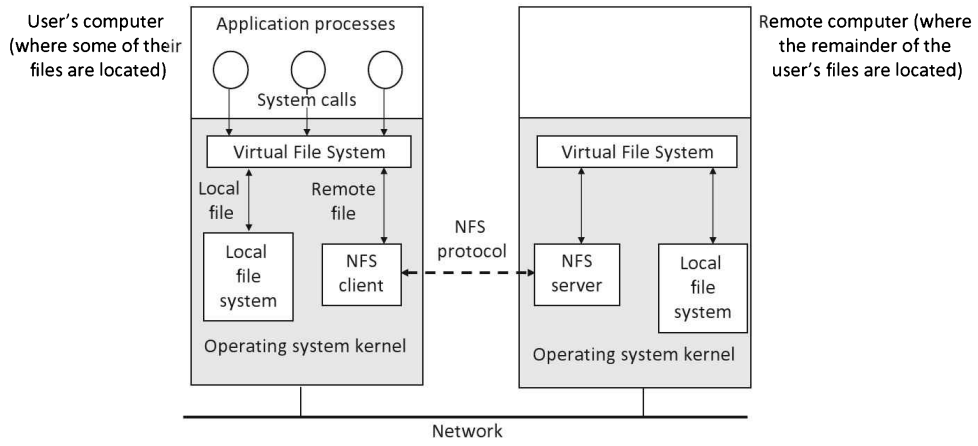


FIGURE 6.2

Access transparency provided by the virtual file system.

is then retrieved and passed back to the user through the VFS layer, such that the user is shielded from the complexity of what has gone on; hence, they just get access to the file they requested.

6.2.2 LOCATION TRANSPARENCY

One of the most commonly occurring challenges in distributed systems is the need to be able to locate resources and services when needed. Some desirable capabilities of distributed systems actually increase the complexity of this challenge: such as the ability to replicate resources, the ability to distribute groups of resources so that they are split over several locations, and the ability to dynamically reconfigure systems and services within, for example, to adapt to changes in workload, to accommodate larger numbers of users, or to mask partial failures within the system.

Location transparency is the ability to access objects without the knowledge of their location. A main facilitator of this is to use a resource naming scheme in which the names of resources are location-independent. The user or application should be able to request a resource by its name only, and the system should be able to translate the name into a unique identifier that can then be mapped onto the current location of the resource.

The provision of location transparency is often achieved through the use of special services whose role is to perform a mapping between a resource's name and its address; this is called name resolution. This particular mechanism is explored in detail later in this chapter in the section that discusses name services and the Domain Name System (DNS).

Resource virtualization using a special layer or service (as discussed in "Access transparency") also provides location transparency. This is because the requesting process does not need to know where the resource is located in the system, it only has to pass a unique identifier to the service, and the service will locate the resource.

Communication mechanisms and protocols that require that the address of the specific destination computer be provided are not location-transparent. Setting up a TCP connection with, or sending a UDP datagram to, a specific destination is clearly not location-transparent (unless the address has

been automatically retrieved in advance using an additional service), but nevertheless, the underlying mechanisms of TCP and UDP (when used in its point-to-point mode) are not location-transparent.

Higher-level communication mechanisms such as remote procedure call (RPC) and Remote Method Invocation (RMI) require that the client (the calling process) knows the location of the server object (this is usually supplied as a parameter to the local invocation). Therefore, RPC and RMI are not location-transparent. However, as with socket-level communication discussed above, these mechanisms can be used in location-transparent applications by prefetching the target address using another service such as a name service.

Middleware systems provide a number of forms of transparency including location transparency. The whole purpose of middleware is to facilitate communication between components in distributed systems while hiding the implementation and distribution aspects of the system. This requires built-in mechanisms to locate components on demand (usually based on a system-wide unique identifier) and to pass requests between components in access and location-transparent ways. An object request broker (ORB) is a core component of middleware that automatically maps requests for objects and their methods to the appropriate objects anywhere in the system. Middleware has been introduced in Chapter 5 and is also discussed in further detail later in this chapter.

Group communication mechanisms (discussed later in this chapter) provide a means of sending a message to a group of processes without knowing the identity or location of the individuals and thus provide location transparency.

Multicast communication also permits sending a message to a group of processes, but there are several different implementations. In the case that the recipients are identified collectively by a single virtual address that represents the group, then it is location-transparent. However, in the case where the sender has to identify the individual recipients in the form of a list of specific addresses, it is not location-transparent.

Broadcast communication is inherently location-transparent as it uses a special address that causes a message to be sent to all possible recipients. A group of processes can communicate without having to know the membership of the group (such as the number of processes and the location of each).

6.2.3 REPLICATION TRANSPARENCY

Distributed systems can comprise many resources and can have many users who need access to those resources. Having a single copy of each resource can be problematic; it can lead to performance bottlenecks if lots of users need to access the same resource and it also leaves the system exposed to the risk of one of the resources becoming unavailable, preventing work being done (e.g., a file becomes corrupted or a service crashes). For these reasons, among others, replication of resources is a commonly used technique.

Replication transparency requires that multiple copies of objects can be created without any effect of the replication seen by applications that use the objects. This implies that it should not be possible for an application to determine the number of replicas or to be able to see the identities of specific replica instances. All copies of a replicated data resource, such as files, should be maintained such that they have the same contents, and thus, any operation applied to one replica must yield the same results as if applied to any other replica. The provision of transparent replication increases availability because the accesses to the resource are spread across the various copies. This is relatively easy to provide for read-only access to resources but is complicated by the need for consistency control where updates to data objects are concerned.

The focus of the following discussion is on the mechanistic aspects of implementing replication. The need for resources to be replicated to achieve nonfunctional requirements in systems, including robustness, availability, and responsiveness, has been discussed in detail in Chapter 5. There is also an activity in that chapter that explores replication in action.

The most significant challenge of implementing replication of data resources is the maintenance of consistency. This must be unconditional; under all possible use scenarios, the data resources must remain consistent. Regardless of what access or update sequence occurs, the system should enforce whatever controls are needed to ensure that all copies of the data remain correct and reflect the same value, such that whichever copy is accessed, the same result is achieved. For example, a pair of seemingly simultaneous accesses to a particular resource (by two different users) may actually be serialized such that one access takes place and completes before the other starts. This should be performed on short timescales so that users do not notice the additional latency of the request and thus have the illusion that they are the only user (see also "Concurrency transparency" for more details).

When one copy of a replicated data resource is updated, it is necessary to propagate the change to the other copies (maintaining consistency as mentioned above) before any further accesses are made to those copies to ensure that out-of-date data are not used and that updates are not lost.

There are several different update strategies that can be used to propagate updates to multiple replicas. The simplest update strategy is to allow write access at only one replica, limiting other replicas to read-only access. Where multiple replicas can be updated, the control becomes more complex; there is a trade-off between the flexibility benefits of allowing multiple replicas to be accessed in a read-write fashion and the additional control and communication overheads of managing the update propagation.

Access to shared resources is usually provided via services. Application processes should not in general access shared resources directly because, in such a case, it is not possible to exercise access control and therefore maintain consistency. Therefore, the replication of data resources generally implies the replication of the service entities that manage the resources. So, for example, if the data content of a database is to be replicated, then there will need to be multiple copies of the database service, through which access to the data is facilitated. Figure 6.3 illustrates some models for server replication and update propagation.

Figure 6.3 shows three different models for server replication. The most important aspect of these models is the way in which updates are performed after a process external to the service has written new data or updated existing data. The fundamental requirement is that all replicas are just that (exact replicas of each other) a service is inconsistent if the different copies of its data do not hold identical values. Part A of the figure shows the primary-backup model (also called the master-backup model). Here, only the primary instance is visible to external processes, so all accesses (read and write) are performed at the primary, and thus, there can be no inconsistency under normal conditions (i.e., when the primary instance is functioning). Updates performed at the primary instance are propagated to the backup instance at regular periods or possibly immediately each time the primary data are changed. The backup instance is thus a "hot spare"; it has a full copy of the service's data and can be made available to external processes as soon as the primary node fails. The strength of this approach is its simplicity, while it has the weaknesses of only providing one instance of the service at any given time (so is not scalable). There is a small window of opportunity for the two copies to become inconsistent, which can arise if the primary updates its database and then crashes before managing to propagate the update to the backup instance. Primary-backup (master-backup) replication has been explored in detail in activity A3 in Chapter 5.

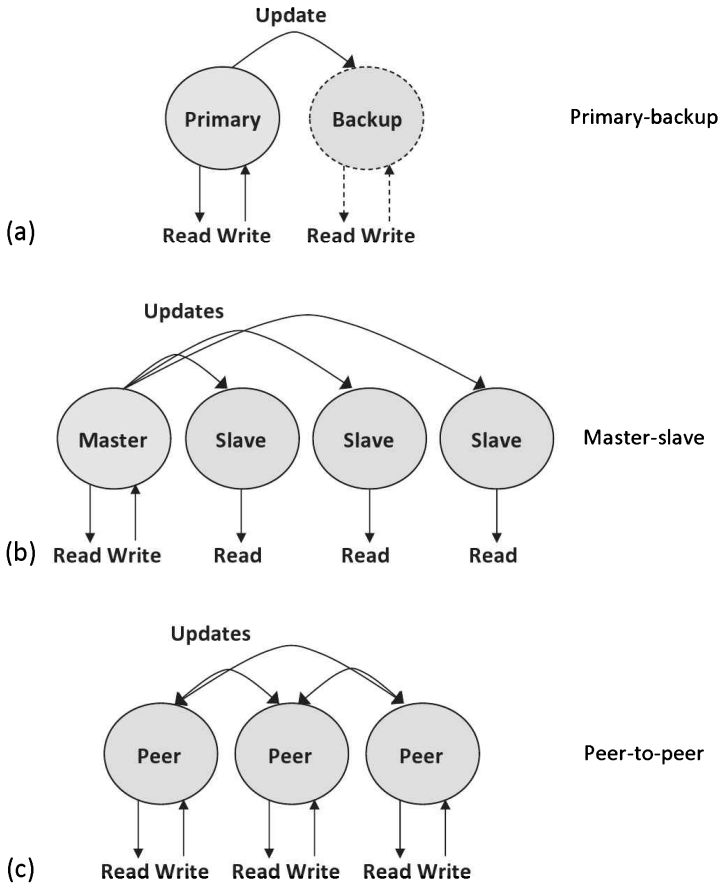


FIGURE 6.3

Some alternative models for server replication.

Figure 6.3 part B shows the master-slave model. All instances of the service can be made available for access by external processes, but write requests are only supported by the master instance and must be propagated to all slave instances as soon as possible so that reads are consistent across the entire service. This replication model is ideal for the large number of applications in which read access is more frequent than write access. For example, in file systems and database systems, reads tend to be more common than writes because updates tend to require a read-modify-write sequence, and thus, writing incorporates a read (the exception being when new files or records, respectively, are created), but reading doesn't incorporate a write. There is no absolute limit to the number of slave instances so the model is very scalable with respect to read requests. A large number of write requests can become a bottleneck because of the requirement to propagate updates, and this becomes more severe as the number of slaves (whose contents must be kept consistent) becomes higher. Mechanisms such as the two-phase commit (see below) should be used to ensure that all updates are either completed (at all nodes) or rolled back to the previous state so that all copies of the data remain consistent even when one instance cannot be updated. In the case of a rollback, the affected update is lost; in which case, the external process that submitted the original request must resubmit it to the service. The slave instances are all potential "hot

spares" for the master, should it fail. An election algorithm (see later) is required to ensure that failure of the master is detected and acted upon quickly, selecting exactly one slave to be promoted to master status. The existence of multiple master instances would violate the consistency of the data as multiple writes could occur at the same data record or file at the same time but at different instances, leading to lost updates and inconsistent values held at different nodes.

Figure 6.3 part C shows the peer-to-peer model. This model requires very careful consideration of the semantics of file updates and the way that the data are replicated. It is challenging to implement where global consistency is required at the same time as requiring that each replica has a full copy of the data. This service can however be very useful where the data are fragmented and each node only holds a small subset of the entire data, and thus, the level of replication is lower (i.e., when an update must be performed, it only has to be propagated to the subset of nodes that have copies of the particular data object). This replication model is ideally suited to applications where most of the data are personalized to particular end users, and thus, there is a naturally low level of replication. This approach has become very popular for mobile computing with applications such as file sharing and social networking running on user's portable computing devices such as mobile phones and tablets.

6.2.3.1 Invalidation

Caching of local copies of data within distributed applications is a specialized form of replication to reduce the number of remote access requests to the same data. Caching schemes tend to not update all cached copies when the master copy is updated; this is because of the communication overhead involved, coupled with the possibility that the process holding the cached copy may not actually need to access the data again (so propagating the update would be wasted effort). In such systems, it is still necessary to inform the cache holders that the data they hold are out of date; this is achieved with an invalidation message. In such case, the application only needs to rerequest the data from the master copy if it needs to access the same again.

6.2.3.2 Two-Phase Commit (2PC) Protocol

Updating multiple copies of a resource simultaneously requires that all copies are successfully updated; otherwise, the system will become inconsistent. For example, consider that there are three copies of a variable named *X*, which has the value 5 initially. An update to change the value to 7 occurs; this requires that all three copies are changed to the value 7. If only two of the values change, the consistency requirement is violated. There could however be reasons why one of the copies cannot be updated: perhaps the network connection has temporarily failed. In such a case where one copy cannot be updated, then none of them should be, once again preserving consistency. In such a situation, the requestor process wanting to perform the update will be informed that the update failed and will be able to resubmit the update later.

The two-phase commit protocol is a well-known technique to ensure that the consistency requirement is met when updating replicas. The first of the two phases determines whether it is possible to update all copies of the resource, and the second phase actually commits the change on an all-or-none basis.

Operation: A commit manager (CM) coordinates a transaction that may involve several participating processes to update replicated data at several sites (see Figure 6.4):

- Phase 1. An update request is sent to each participating process and each replies with an acknowledgment that they have managed to perform the update (or otherwise that they are not

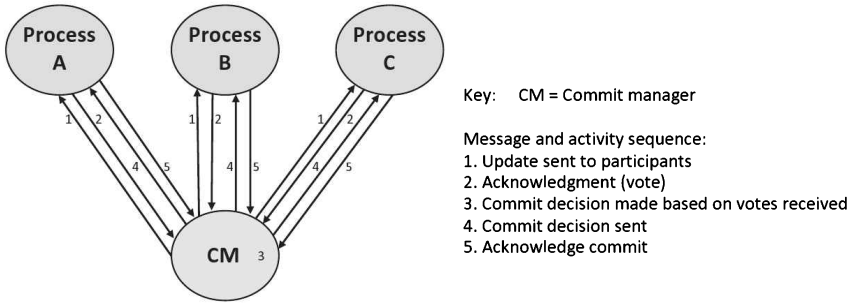


FIGURE 6.4

The two-phase commit protocol.

able to). Updates are not made permanent at this stage; a rollback to the original state may be required. The acknowledgments serve as **yes** or **no** votes.

- Phase 2. The CM decides whether to commit or abort based on the votes received. If all processes voted yes in phase 1, then a commit decision will be taken; otherwise, the transaction is aborted. The CM then informs participating nodes whether to commit the transaction (i.e., make the changes permanent or rollback).

Figure 6.4 shows the behavior and sequence of messages that constitute the two-phase commit protocol.

The first two messages 1 and 2 represent the first phase of the protocol in which the update is sent to each participating process and they send their votes back (informing of their ability to perform the update). The CM then decides whether or not to commit, based on the votes (step 3). The final two messages 4 and 5 represent the second phase in which the processes are told whether to commit or abort, and each sends back an acknowledgment to confirm their compliance.

6.2.4 CONCURRENCY TRANSPARENCY

Distributed systems can comprise many shared-access resources and can have many users (and applications running on behalf of users), which use those resources. The behavior of users is naturally asynchronous; this means that they each perform actions when they need to without knowing or checking what others are doing. The resulting asynchronous nature of resource accesses means that there will be occasions where two or more processes attempt to access a particular resource simultaneously.

Concurrency transparency requires that concurrent processes can share objects without interference. This means that the system should provide each user with the illusion that they have exclusive access to the resource.

Concurrent access to data objects raises the issue of data consistency, but from a slightly different angle to that discussed in the context of data replication (above). In the case of replication, there are multiple copies of a resource being updated with the same value, whereas with concurrent access, there are multiple entities updating a single resource. These are different variations of the same fundamental problem and equally important.

In a situation where two or more concurrent processes attempt to access the same resource, there needs to be some regulation of actions. If both processes only read the resource data, then the order of

the two reads does not matter. However, if one or both of them write a new value to the resource, then the sequence with which the accesses take place becomes critical to ensuring that data consistency is maintained. Typically, updating a data value follows a read-update-write sequence. If each whole sequence is isolated from other sequences (e.g., by wrapping them inside a transaction mechanism or by locking the resource for the duration of the sequence so that one process has temporary exclusive access), then consistency is preserved. However, where the access sequences of the two processes are allowed to become interleaved, the lost update problem can arise and the system becomes inconsistent (the lost update problem was introduced in Chapter 4 and is discussed further below).

Figure 6.5 illustrates the lost update problem, using an airline booking system as a case example. The application must support multiple concurrent users who are unaware of the existence of other users. The users should be presented with a consistent view of the system, and even more importantly, the underlying data stored in the system must remain consistent at all times. This can be difficult to achieve in highly dynamic applications. For the airline booking application, we consider the consistency requirement that (for a specific flight, on a specific aircraft) the total number of seats available plus the total number of seats booked must always be equal to the total number of seats on the aircraft. It must not be possible for two users to manage to book the same seat or for seats to be "lost," for example, a seat is booked and then released, but somehow is not added back to the available pool.

The scenario shown in Figure 6.5 starts in a consistent state in which there are 170 seats available. In step 1, client1 reads the number of seats available and caches it locally. Soon after this, in step 2, client2 does the same. Imagine that this activity is somehow linked to users browsing an online booking system, taking a while to decide whether to book or not and then submitting their order. Client1 then books 2 seats and writes back the new availability, which is $170-2=168$ (step 3 in the figure). Later, client2 books 3 seats and writes back the new availability, which is $170-3=167$ (step 4). The true availability of seats is now $170-(2+3)=165$. The sequence of accesses in the scenario has led to the creation of two seats in the system that do not actually exist on the plane; therefore, the system is inconsistent.

It is important that you can see where the problem lies in this scenario: it arises because the sequences of accesses of the two clients were allowed to overlap. If client2 had been forced to reread the availability of seats after client1's update, then the system would have remained consistent.

In addition to the consistency aspect of concurrency transparency, there is also a performance aspect. Ideally, user requests can be interleaved on a sufficiently fine-grained timescale that they do not notice a performance penalty arising from the forced serialization occurring behind the scenes. If

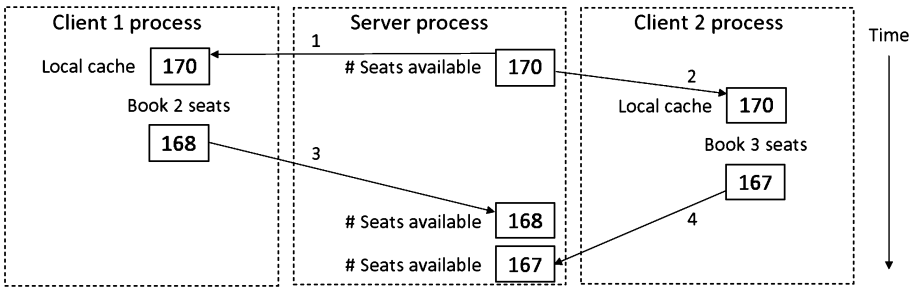


FIGURE 6.5

The lost update problem (illustrated with an airline seat booking scenario).

resources are locked for long periods of time, the concurrency transparency is lost, as users notice that transaction times increase significantly when system load or the number of users increases.

Important design considerations include deciding what must be locked and when and ensuring that locks are released promptly when no longer needed. Also important is the scope of the locking; it is undesirable to lock a whole group of resources when only one item in the group is actually accessed. To put this into context, consider the options for preserving database consistency: locking entire databases during transactions temporarily prevents access to the entire database and is highly undesirable from a concurrency viewpoint. Table-level locking enables different processes to access different tables at the same time, because their updates do not interfere, but still prevents access to the entire locked table even if only a single row is being accessed. Row-level locking is a fine-grained approach that increases transparency by allowing concurrent access at the table level.

6.2.4.1 Transactions

A transaction is an indivisible sequence of related operations that must be processed in its entirety or aborted without making any changes to system state. A transaction must not be partially processed because that could corrupt data resources or lead to inconsistent states.

Transactions were introduced in Chapter 4. Recall that there are four mandatory properties of transactions (the ACID properties):

- Atomicity. The transaction cannot be divided. The whole transaction (all suboperations) is carried out or none of it is carried out.
- Consistency. The stored data in the system must be left in a consistent state at the end of the transaction (this leads on from the all-or-nothing requirement of atomicity). Achieving the requirement of consistency is more complex if the system supports data replication.
- Isolation. There must be no interference between transactions. Some resources need to be locked for the duration of the transaction to ensure this requirement is met. Some code sections need to be protected so that they can only be entered by one process at a time (the protection mechanisms are called MUTEXs because they enforce mutual exclusion).
- Durability. The results of a transaction must be made permanent (stored in some nonvolatile storage).

Figure 6.6 relates together the four properties of transactions in the context of a multipart transaction in which partial results are generated but must not be made visible to processes outside of the transaction.

The transaction properties are placed into the perspective of a banking application example:

The banking application maintains a variety of different types of data, in a number of databases. These data could include customer accounts data, financial products data (e.g., the rules for opening different types of accounts and depositing and withdrawal of funds), interest rates applicable, and tax rates applicable. A variety of different transactions can occur, depending on circumstances. Transaction types could include open new account, deposit funds, withdraw funds, add net interest, generate annual interest and tax statement, and close account. Each of these transactions may be internally subdivided into a number of steps and may need to access one or more of the databases.

Consider a specific transaction type: add net interest. This transaction may need to perform the following steps: (1) read the account balance and multiply by the current gross interest rate to determine the gross interest amount payable; (2) multiply the gross interest amount by the tax rate to determine

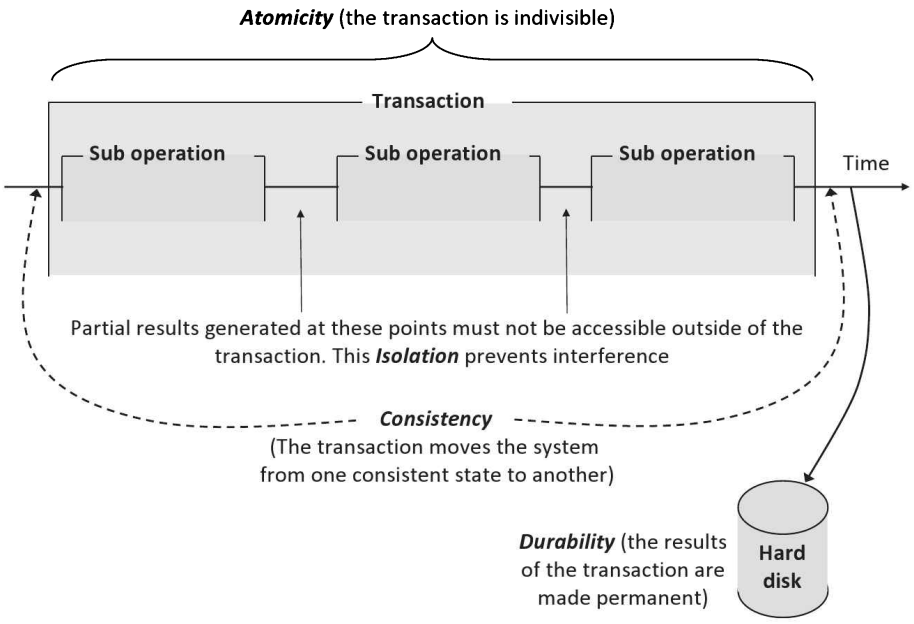


FIGURE 6.6

The four properties of transactions.

the tax payable on the interest; (3) subtract the tax payable from the gross interest amount to determine the net interest payable.

Prior to the transaction, the system is in a consistent state and the following three values are stored (in three separate databases): account balance = £1000; interest rate = 2%; tax rate = 20%. Figure 6.7 illustrates the internal operation of the transaction.

Figure 6.7 places the ACID transaction properties into the perspective of the mechanics of the banking application transaction. The transaction operates in three steps leading to temporary internal states (partial results), which must not be visible externally. When the transaction completes, the system is in a new consistent state, the account balance having been updated to the value £1016.

The isolation property of transactions is particularly important with respect to concurrency transparency because it prevents external processes accessing partial results generated temporarily as a transaction progresses. In the example illustrated in Figure 6.7, a temporary value of £1020 is written to the account, but the transaction is only partially complete at this stage, and tax is yet to be deducted. Therefore, the user never actually has that amount to withdraw. If the value were exposed to other processes, then the system could become inconsistent. If the user were to check their balance in the short time window while the temporary value was showing on the account, they would think they had more money than they actually do have. Worse still would be if the user were allowed to withdraw £1020 at this point because they do not actually have that much money in the account. When the transaction finally completes, the system is left in a consistent state with £1016 in the account; this value can now be exposed to other processes.

A further example is provided by re-presenting the airline seat booking system (seen earlier in this section) as a series of transactions. The forced serialization (and thus isolation) overcomes the inconsistency weakness of the earlier design.

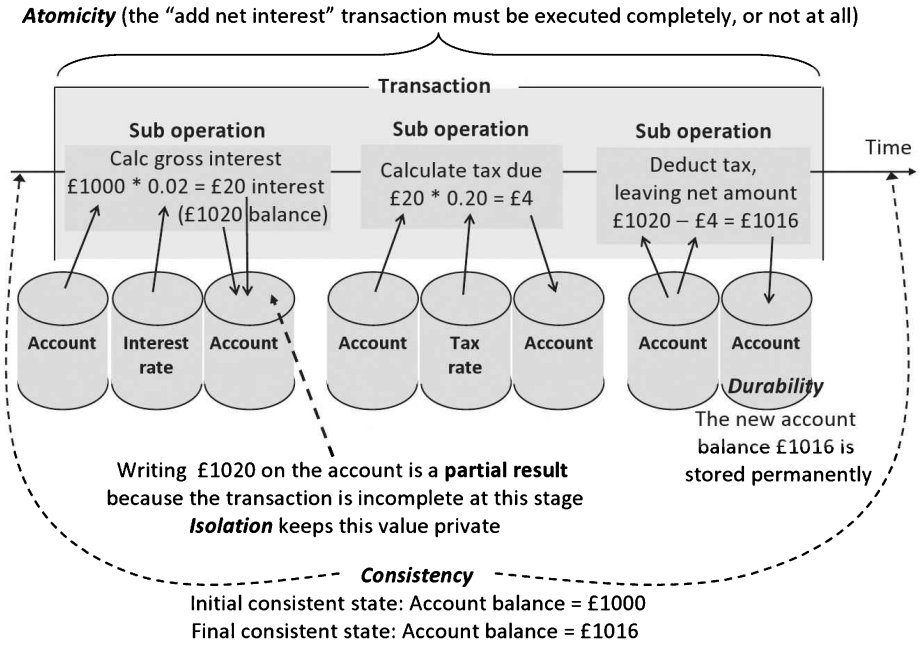


FIGURE 6.7

The internal operation of the "add net interest" transaction of the banking application.

Figure 6.8 presents a transaction implementation of the airline seat booking system. This approach serializes the two seat booking operations that would lead to an inconsistent state if allowed to become interleaved. The system is left in a consistent state after each transaction, in which the number of seats available and the number of seats booked always are equal to the total number of seats on the aircraft.

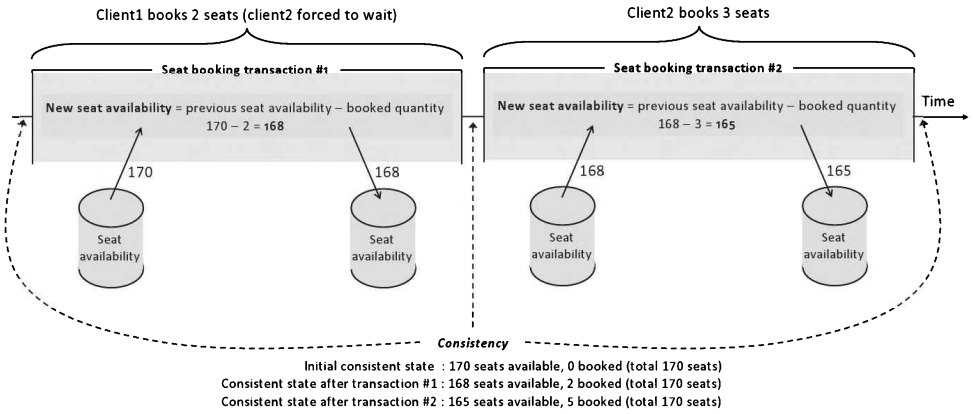


FIGURE 6.8

A transaction implementation of the airline seat booking system.

6.2.5 MIGRATION TRANSPARENCY

Distributed systems tend to be dynamic in a number of ways including changes in the user population and the activities they carry out, which in turn leads to load-level fluctuations on different services at various times. Systems are also dynamic due to the addition and relocation of physical computers, continuous changes in network traffic levels, and the random failure of computers and network connections.

Due to the dynamic nature of these systems, it is necessary to be able to reconfigure the resources within the system, for example, to relocate a particular data resource or service from one computer to another.

Migration transparency requires that data objects can be moved without affecting the operation of applications that use those objects and that processes can be moved without affecting their operations or results.

In the case of migration of data objects (such as files) in active use, access requests from processes using the object must be transparently redirected to the object's new location. Where objects are moved between accesses, techniques used to implement location transparency can suffice. A name service (discussed later) provides the current location of a resource; its history of movement is of no consequence as long as its current location after moving is known. However, there is a challenge in keeping the name service itself up to date if resources are moved frequently.

Process transfers can be achieved preemptively or nonpreemptively. Preemptive transfers involve moving a process in midexecution. This is complex because the process' execution state and environment must be preserved during the transfer. Nonpreemptive transfers are done before the task starts to execute so the transfer is much simpler to achieve.

6.2.6 FAILURE TRANSPARENCY

Failures in distributed systems are inevitable. There can be large numbers of hardware and software components interacting, dependent on the communication links between them. The set of possible configurations and behaviors that can arise is too great in general that every scenario can be tested for; therefore, there will always be the possibility of some unforeseen combination of circumstances that leads to failure. In addition to any built-in reliability weaknesses they may have, hardware devices and network links can fail due to external factors; for example, a power cut may occur or a cable is accidentally unplugged.

While it is not possible to prevent failures outright, measures should be taken to minimize the probability of failures occurring and to limit the consequences of failures when they do happen.

Good system-level design should take into account the reality that any component can fail and should avoid having critical single points of failure where possible. Software design should avoid unnecessary complexity within components and in the connectivity with, and thus dependency upon, other components (see the section on component coupling in Chapter 5). Additional complexity increases the opportunities for failure; and the more complex the system, the harder it is to test. This leads to undertesting where not all of the functional and behavioral scope is covered by tests. Even where testing is thorough, it cannot prove that a failure will not take place. Latent faults are present in most software systems. These are faults that have not yet shown up; faults in this category often only occur in a particular sequence or combination of events so they can lurk undetected for a long time.

Once we have exhausted all possible design-time ways to build our systems to be as reliable as possible, we need to rely on runtime techniques that will deal with the residual failures that can occur.

Failure transparency requires that faults are concealed so that applications can continue to function without any impact on behavior or correctness. Failure transparency is a significant challenge! As mentioned above, many different types of fault can occur in a distributed system.

Communication protocols provide a good example of how different levels of runtime failure transparency can be built in through clever design. For example, compare the TCP and UDP. TCP has a number of built-in features including sequence numbers, acknowledgments, and a retransmission-on-time-out mechanism that transparently deal with various issues that occur at the message transmission level, such as message loss, message corruption, and acknowledgment loss. UDP is a more lightweight protocol and, as a result, has none of these facilities and, hence, is commonly referred to as a "send and pray" protocol.

6.2.6.1 Support for Failure Transparency

A popular technique to provide a high degree of failure transparency is to replicate processes or data resources at several computing hosts, thus avoiding the occurrence of a single point of failure (see "Replication transparency").

Election algorithms can be used to mask the failure of critical or centralized components. This approach is popular in services that need a coordinator and can include replicated services in which one copy is allocated the role of master or coordinator. On the failure of the coordinator, another service member process will be elected to take over the role. Election algorithms are explored in detail later in this chapter.

In all replicated services or situations where a new coordinator is elected when a failure occurs, the extent to which the original failure is masked is dependent on the internal design of the service, in particular the way in which state is managed. The new coordinator may not have a perfect copy of the state information that the previous one had (e.g., an instantaneous crash can occur during a transaction), so the system may not be in an identical situation after recovery. This scenario should be given careful consideration during the design effort to maximize the likelihood that the system does remain entirely consistent across the handover. In particular, the use of stateless server design can reduce or remove the risk of state loss on server failure, as all state is held on the client side of connections (see the discussion on stateful versus stateless services in Chapter 5).

Even when stateless services are used, problems can still arise. For example, if a single requested action is carried out multiple times at a server, the correctness or consistency of the system could be disrupted. Consider, for example, in a banking application, the function "Add annual interest" being executed multiple times by accident. This could happen where a request is sent and not acknowledged; the client may resend the message on the assumption that the original request was lost, but in fact, the message had arrived and it was actually the acknowledgment that was lost; the outcome being that the server will receive two copies of the request. One way to resolve this is to use sequence numbers so that duplicate requests can be identified. Another approach is to design all actions to be idempotent.

An idempotent action is one that is repeatable without having side effects. The use of idempotent requests contributes to failure transparency because it hides the fact that a request has been repeated, such that the intended outcome is correct. There is no side effect of requesting the action more than once, whether the action is actually carried out multiple times or not. Another way to think of this is that the use of idempotent actions allows certain types of fault to occur while preventing them from having any impact and therefore removing the need to handle the faults or recover from them.

Generic request types that can be idempotent include "set value to *x*," "get value of item whose ID is *x*," and "delete item whose ID is *x*."

Generic request types that are not idempotent include "add value *y* to value *x*," "get next item," and "delete item at position *z* in the list."

A specific example of a nonidempotent action is "add 10 pounds to account number 123." This is because (if we assume the initial balance is 200 pounds) after being executed once, the new balance will be 210 pounds, and after two executions, the balance will be 220 pounds and so on.

However, the action can be reconstructed as a series of two idempotent actions, each of which can be repeated without corrupting the system's data. The first of the new actions is "get account balance," which copies the balance from the server side to the client side. The result of this is that the client is informed that the balance is 200 pounds. If this request is repeated multiple times, the client is informed of the balance value several times, but it is still 200 pounds. Once the client has the balance, it then locally adds 10 pounds. The second idempotent action is "set account balance to 210 pounds." If this action is performed once or more times in succession, the balance at the server will always end up at 210 pounds. In addition to being robust, the approach of shifting the computation into the client further improves the scalability of the stateless server approach. However, this technique is mostly useful for nonshared data, such as the situation in the example above in which each client is likely to be interested in a unique bank account. Where the data are shared, as in the earlier airline seat booking example, transactions (which are more heavyweight and less scalable) are more appropriate due to the need for consistency across multiple actions and the need to serialize accesses to the system data.

Idempotent actions are also very useful in safety-critical systems, as well as in systems that have unreliable communication links or very high latency communication. This is because the role of acknowledgments is less critical when idempotent requests are used and the not uncommon problem of the time-out being too short (causing retransmission) does not result in erroneous behavior at the application level.

Checkpointing is an approach to fault tolerance that can be used to protect from the failure of critical processes or from failure of the physical computers that critical processes run on.

Checkpointing is the mechanism of making a copy of the process' state at regular intervals and sending that state to a remote store (the state information that is stored includes the process' memory image, such as the value of variables, as well as process management details such as which instruction is to be executed next, and IO details such as which files are open and what communication links are set up with other processes.

If a checkpointed process crashes or its host computer fails, then a new copy of the process can be restarted using the stored state image. The new process begins operating from the point the previous one was at when the last checkpoint was taken. This technique is particularly valuable for protecting the work performed by long-running processes such as occur in scientific computing and simulations such as weather forecasts that may run for many hours. Without checkpointing, a failed process would have to start again from the beginning, potentially causing the loss of a lot of work and causing delay to the user who is waiting for the result.

6.2.7 SCALING TRANSPARENCY

For distributed systems in general, as the system is scaled up, a point is eventually reached where the performance will begin to drop; this could be noticed, for example, in terms of slower response times or service requests timing-out. Small increases in scale beyond this point can have severe effects on performance.

Scaling transparency requires that it should be possible to scale up an application, service, or system without changing the system structure or algorithms. Scaling transparency is largely dependent on efficient design, in terms of the use of resources, and especially in terms of the intensity of communication.

Centralized components are often problematic for scalability. These can become performance bottlenecks as the number of clients or service requests increases. Centralized data structures grow with system size, eventually causing scaling problems in terms of the size of the data and the increased time taken to search the larger structure, impacting on service response times. A backlog of requests can build up rapidly once a certain size threshold is exceeded.

Distributed services avoid the bottleneck pitfalls of centralization but represent a trade-off in terms of higher total communication requirements. This is because in addition to the external communication between clients and the service, there is also the service-internal communication necessary to coordinate the service and, for example, to propagate updates between server instances.

Hierarchical design improves scalability; a very good example of this is the DNS that is discussed in detail later in this chapter. Decoupling of components also improves scalability. Publish-subscribe event notification services (also discussed later) are an example technique by which the extent of coupling and also the communication intensity can be significantly reduced.

6.2.7.1 Communication Intensity: Impact on Scalability

Interaction complexity is a measure of the number of communication relationships within a group of components. Because systems scale can change, interaction complexity is described in terms of the proportion of the other components in the system each component communicates with, rather than absolute numbers (see Table 6.1).

Communication intensity is the amount of communication that actually occurs, which results from the interaction complexity combined with the actual frequency of sending messages between communicating components and the size of those messages. This is a significant factor that limits the scalability

Table 6.1 Example Interaction Complexities for a System of N Components

Typical proportion of other components that each of the N components communicates with	Interaction complexity	Typical interpretation
1	$O(N)$	Each component communicates with another component. This is highly scalable as the communication intensity increases linearly as the system size increases
2	$O(2N)$	Each component communicates with two other components. Communication intensity increases linearly as the system size increases
$N/2$	$O(N^2/2)$	Each component communicates with approximately half of the system. This represents an exponential rate of increase of communication intensity and thus can impact on scalability
$N-1$	$O(N^2-N)$ also written $O(N(N-1))$	Each component communicates with most or all. This is a steep exponential relationship and can severely impact on scalability

of many systems. This is because the communication bandwidth is finite in any system and the communication channels become bottlenecks as the amount of communication builds up. Communication is also relatively time-consuming compared with computation. As the ratio of time spent communicating (including waiting to communicate due to network congestion and access latency) to computation time increases, the throughput and efficiency of systems fall. The reduction in performance eventually becomes a limiting factor on usability. If the performance cannot be restored by adding more resource while leaving the design unchanged, then the design is said to be not scalable or to have reached the limit of its scalability.

Some diverse examples of interaction complexity are provided below:

- The bully election algorithm in its worst-case scenario during an election has $O(N^2 - N)$ interaction complexity (the bully election algorithm is discussed in "Election algorithms" later in this chapter).
- A peer-to-peer media sharing application (as explored in activity A2 in Chapter 5) may have typical interaction complexity of between $O(2N)$ and $O(N^2/2)$ depending on the actual proportion of other peers that each one connects to.
- The case study game that has been used as a common point of reference throughout the book has a very low interaction complexity. Each client connects to one server component regardless of the size of the system, so system-wide interaction complexity is $O(N)$ where N is the number of clients in the system.

Figure 6.9 shows two different interaction mappings between the components of similar systems. Part A illustrates a low-intensity mapping in which each component connects with on average one other, so the interaction complexity is $O(N)$. In contrast, part B shows a system of highly coupled components in which each component communicates with approximately half of the others, so the interaction complexity is $O(N^2/2)$.

Figure 6.10 provides a graphic illustration of the way in which interaction complexity affects the relationship between the size of the system (the number of components) and the resulting intensity of communication (the number of interaction relationships). The steepness of the curves associated with the more intense interaction complexities illustrates the relative severity of their effect on scalability.

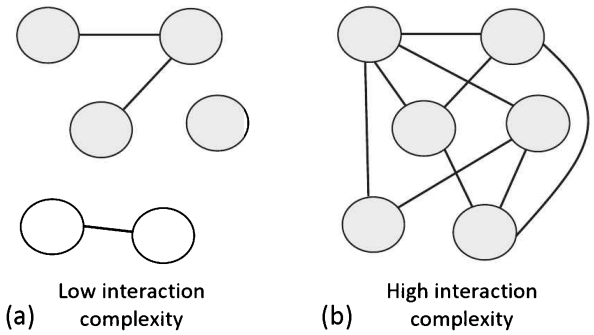


FIGURE 6.9

Illustration of low and high interaction complexity.

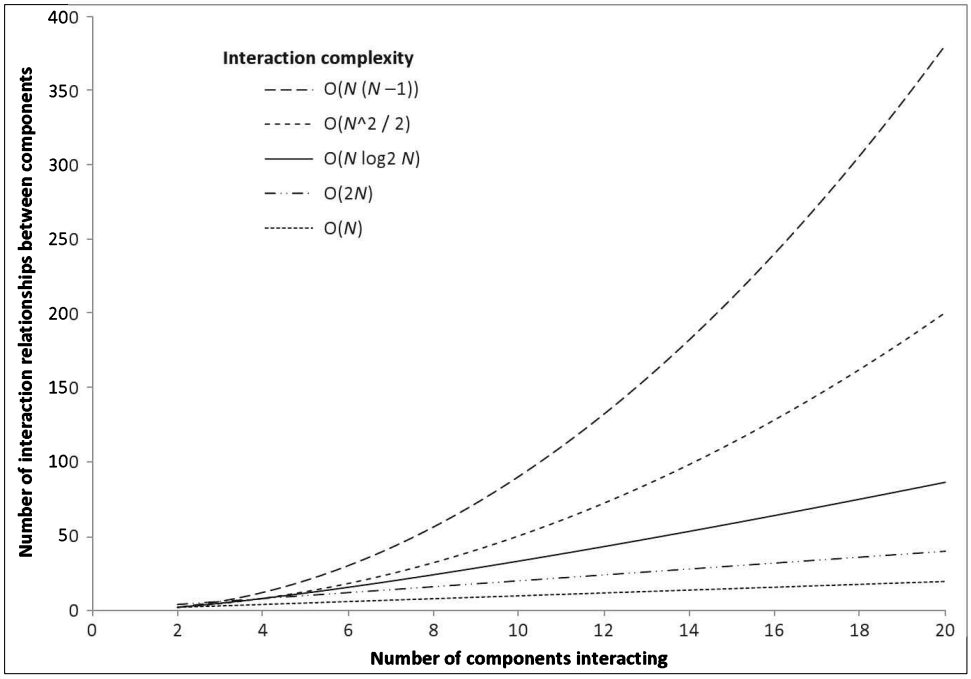


FIGURE 6.10

The effect of various interaction complexities on the relationship between the size of the system and the resulting intensity of communication.

6.2.8 PERFORMANCE TRANSPARENCY

The performance of distributed systems is affected by numerous aspects of their configuration and use.

Performance transparency requires that the performance of systems should degrade gracefully as the load on the system increases. Consistency of performance is a significant aspect of the user experience and can be more important than absolute performance. A system that has consistent good performance is better received than one in which the performance is outstanding some of the time but can degrade rapidly and unpredictably, which leads to user frustration. Ultimately, this is a measure of usability.

6.2.8.1 Support for Performance Transparency

Performance (an attribute of the system) and performance transparency (a requirement on performance) are affected by the design of every component in the system and also by the collective behavior of the system that itself cannot be predicted by knowing the behavior of each individual component, due to the complex runtime relationships and sequences of events that occur.

High performance cannot therefore be guaranteed through the implementation of any particular mechanism; rather, it is an emergent characteristic that arises through consistently good design technique across all aspects of the system. Performance transparency is an explicit goal of load-sharing schemes that attempt to evenly distribute the processing load across the processing resources so as to maintain responsiveness.

6.2.9 DISTRIBUTION TRANSPARENCY

Distribution transparency requires that all details of the network and the physical separation of components are hidden such that application components operate as though they are all local to each other (i.e., running on the same computer) and therefore do not need to be concerned with network connections and addresses.

A good example is provided by middleware. A virtual layer is created across the system that decouples processes from their underlying platforms. All communication between processes passes through the middleware in access-transparent and location-transparent ways, providing the overall effect of hiding the network and the distribution of the components.

6.2.10 IMPLEMENTATION TRANSPARENCY

Implementation transparency means hiding the details of the ways in which components are implemented. For example, this can include enabling applications to comprise components developed in different languages; in which case, it is necessary to ensure that the semantics of communication, such as in method calls, are preserved when these components interoperate.

Middleware such as CORBA provides implementation transparency. It uses a special interface definition language (IDL) that represents method calls in a programming language-neutral way such that the semantics of the method call between a pair of components are preserved (including the number parameters and data type of each parameter value and the direction of each parameter, i.e, being passed into the method or returned from the method) regardless of the combination of languages the two components are written in.

6.3 COMMON SERVICES

Distributed applications have a number of common requirements that arise specifically because of their distributed nature and of the dynamic nature of the system and platforms they operate on. Common requirements of distributed applications include

- an automatic means of locating services and resources,
- an automatic means of synchronizing clocks,
- an automatic means of selecting a coordinator process from a group of candidates,
- mechanisms for the management of distributed transactions to ensure consistency is maintained,
- communications support for components operating in groups,
- mechanisms to support indirect and loose coupling of components to improve scalability and robustness.

It is therefore sensible that a group of support services are provided generically in systems, which provide services to applications in standard ways. Application developers can integrate calls to these services into their applications instead of having to implement additional functionality within each application. This saves a lot of duplicated effort that would be costly, would significantly extend lead times, and could ultimately reduce quality as each developer would implement different variations of services leading to possible inconsistency.

In addition to specific functionalities such as those mentioned above, common services also contribute to the provision of all transparency forms discussed earlier in this chapter and also to the nonfunctional requirements of distributed applications discussed in Chapter 5.

6.11 MIDDLEWARE EXAMPLES AND SUPPORT TECHNOLOGIES

6.11.1 THE COMMON OBJECT REQUEST BROKER ARCHITECTURE (CORBA)

There are many types of middleware; CORBA is a standard defined by the Object Management Group (OMG) and is one of the best-known and historically most widely used examples. CORBA has been around since 1991 and is therefore considered to be a legacy technology by some practitioners, in some opinions superseded by other technologies such as Web services (see later). However, it still has many strengths and is still is in use.

CORBA serves as a useful reference model by which to explain some of the mechanics of middleware operation and some of the transparency aspects of middleware.

6.11.1.1 Motivation for CORBA

Distributed systems represent highly dynamic environments in which significant complexity stems from a wide variety of sources, some of which are inherent in the nature of distribution. Complexity challenges include a large location space in which it is necessary to find resources; communication latency; message loss; partial failures of networks; service partitioning (e.g., the functional split between client and server); replication, concurrency, and consistency control; and the requirement of consistent ordering of distributed events. In addition to these inherent forms of complexity, there are some accidental or avoidable forms of complexity that include the continuous rediscovery and reinvention of core concepts and components and lack of a single common design or development methodology. Distributed systems are also subject to various forms of heterogeneity that have been discussed in detail in Chapter 5.

CORBA was created as an answer to the problems of complexity and heterogeneity in distributed systems. With no consensus ever likely on hardware platforms, operating systems, programming languages, or application formats, the goal of CORBA was to facilitate universal interoperability. In this regard, CORBA allows application components to be accessed from anywhere in a network, regardless of the operating system they are running on and the language they are written in.

In CORBA systems, the communicating entities are objects (which are themselves part of larger applications). Dividing into separate entities at the object level allows for flexible fine-grained distribution of services, functionality, data resources, and workload, across the processing resources of the system.

The ORB is the central component of CORBA. An ORB provides mechanisms to invoke methods on local and remote objects, as illustrated in Figure 6.39.

Figure 6.39 shows a simplified representation of a client object making two calls to methods on other objects, showing only the ORB aspects and not other components of the middleware. The client object does not know where the host objects of these methods (the implementation objects) are located. The figure shows that it does not matter whether the implementation objects are local or remote to the client; the ORB handles the method invocation transparently.

As indicated by the example in Figure 6.39, the ORB provides several forms of transparency. When an object (the client) makes a method request, the ORB automatically locates the host object and calls the requested method. The middleware layer hides platform and operating system heterogeneity that may be present when the called object is remote (the so-called object may be hosted on a physical platform of a different hardware type and/or running a different operating system to the client object's platform). The middleware uses a special IDL format when passing arguments to methods, which means that the objects can be written in different languages that use different call semantics; this will not affect the method call.

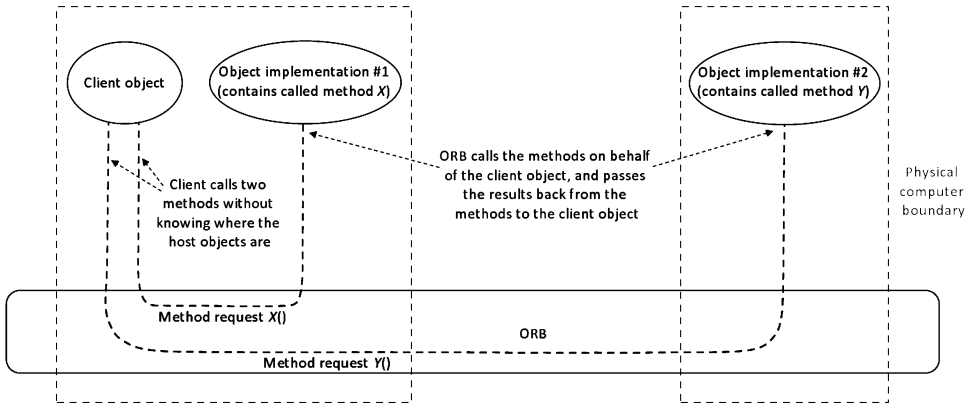


FIGURE 6.39

The object request broker provides a transparent connectivity service supporting method calls.

The transparency provided by the ORB (achieved with support from other components of CORBA) is summarized as follows:

- Access transparency. The calling object does not need to know if a called object is local or remote. Object methods are invoked in the same manner whether they are local or remote to the calling object.
- Location transparency. The location of a called object is not known and does not need to be known by the client object when invoking that object's methods. The called object does not need to know the location of the calling object when passing back results.
- Implementation transparency. Objects interoperate together despite possibly being written in different languages or residing in heterogeneous environments with different hardware platforms or operating systems.
- Distribution transparency. The communications network is hidden. The middleware takes care of setting up low-level (transport-layer) connections between the involved computers and manages all object-level communication using its own special protocols that sit above TCP. At the object-level, a single platform illusion is provided such that objects see all other objects as if they are local.

Figure 6.39 hides a lot of detail as to the actual architecture of CORBA and the actual mechanisms used to invoke method calls. In addition to the ORB, there are several other components and interfaces as shown in Figure 6.40.

Figure 6.40 shows the architecture of CORBA, from the perspective of the ORB and its support components. Also shown are the various interfaces, which are in two categories: those between internal components and also the external interfaces to application objects. Two alternative forms of method invocation are supported—static and dynamic—which each involve different groups of components; the alternative paths are represented by the dotted arrows in the figure.

The subsections below provide details of the roles and behavior of the various components of CORBA and the interactions that occur between the components.

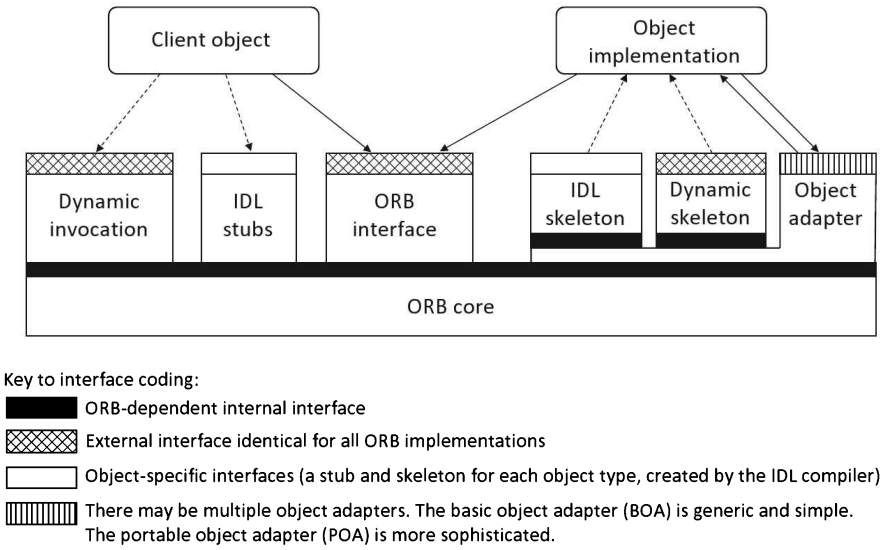


FIGURE 6.40

The ORB-related components and interfaces.

6.11.1.2 Object Adapter and Skeleton

An object adapter is a special component used to interface the CORBA mechanisms to an implementation object (i.e., an object on which a method is to be called). In addition to actually invoking the requested method, the object adapter performs some preparatory actions that include the following:

- Implementation activation and deactivation. The requested application component (the one hosting the object that contains the called method) may not be running (instantiated) when a request arrives to the ORB. If this were to happen in nonmiddleware-based applications in which a client component attempts to connect to a nonrunning server, the call would fail. With CORBA, there is an opportunity for the object adapter to actually instantiate the service component in such circumstances, prior to actually making the method request. It is also possible to leave the application component running after the method has been called or for the object adapter to deactivate it, depending on the server activation policy (see below).
- Mapping object references to implementations. A mapping of which object IDs are provided by each instantiated application component is maintained.
- Registration of implementations. A mapping of the physical location of instantiated application components is maintained.
- Generation and interpretation of object references. When a request to invoke a particular method is received, the ORB locates the particular service instance needed. It does this by relating together the mapping of the required object ID to an instantiation of a specific component and the mapping of instantiated components to locations.
- Object activation and deactivation. Once an object's service host component is instantiated, it may be necessary to separately instantiate specific objects (essentially by creating an instance of the object and calling its constructor).

Once all preparatory steps have been completed, the requested method is invoked (called), with the aid of the object's skeleton.

The skeleton is a server-side object-specific interface that actually performs the method invocation. The skeleton performs three functions: (1) It preserves the semantics of the object request and of the response, by translating the arguments from the IDL representation into the necessary object representation (which is dependent on the implementation language of the server object) and back into IDL for the response; (2) it makes the remote call appear as a local call from the point of view of the server object; (3) it takes care of the network connectivity aspects of the call so that the server application logic need not do so.

The combined operation of the object adapter and the skeleton is portrayed in an example application context in Figure 6.41.

Figure 6.41 illustrates the object adapter's management of method invocation, including the usage of the skeleton, placed into the context of a robot arm application example. In the scenario shown, the client object makes a request to open the robot arm's gripper, by calling the open-gripper method, without knowing the activation status of the server implementation component that hosts the object that contains this method. In the scenario, the server implementation is not already instantiated so the object adapter must first activate (instantiate) an instance of the application component that contains the Robot-Arm-Control object (step 1 in the figure). In step 2, the newly started component is now registered in a repository so that future service requests can be directed to it (otherwise, each new request could result in the instantiation of a dedicated component instance). In step 3, the Robot-Arm-Control object is activated (essentially, this means that an actual object instance of the appropriate class is created and its constructor called). In step 4, the open-gripper method is called using the precreated custom skeleton interface for the object (the skeleton actually performs a local method call on behalf of the remote client object). The server object responds by passing the result of the call back to the skeleton as though the skeleton were the client. This is an important aspect of distribution transparency from the application developer's viewpoint: the fact that the server object thinks it is being called locally means that no special communication considerations need to be made by the server's developer.

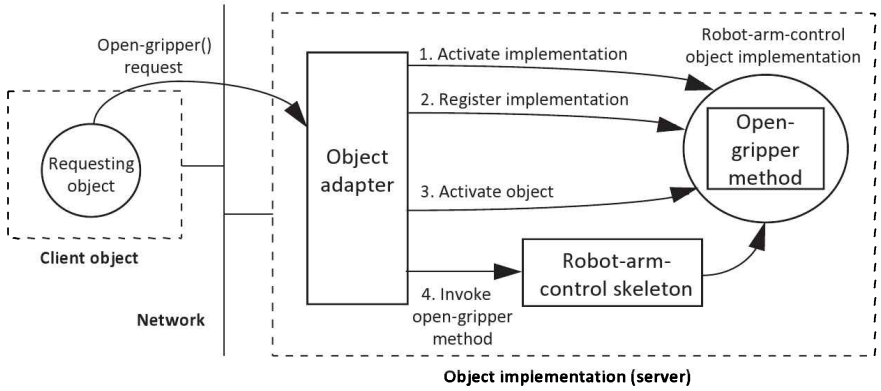


FIGURE 6.41

Object adapter management of method invocation.

6.11.1.3 CORBA Object Servers and Activation Policy

There are several types of object server that can be implemented in CORBA, differentiated by the activation policy that describes the ways the object implementation and object activation are achieved. The different activation policies are as follows:

- Shared server policy. An object adapter activates a given server process the first time that one of its objects receives a request. The server then remains active. A server may implement multiple objects.
- Unshared server policy. The same as a shared server policy except that a server may implement only one object.
- Per-method server policy. Each method request causes a new server process to be started dynamically. Servers terminate once they have executed a method.
- Persistent server policy. Servers are initiated outside of the object adapter. The object adapter is still used to route method requests to the servers. In this situation, if a method request occurs for an object whose server is not already running, the call will fail.

6.11.1.4 CORBA's Repositories

CORBA uses two databases to keep track of the state of the system:

- The implementation repository is used by the ORB and object adapter to keep track of object servers and their runtime instances. In other words, it is a dynamic database of whether or not server applications are actually instantiated (running) and the location details of those that are running, so that requests can be forwarded to them.
- The interface repository is a database that contains descriptions of the methods associated with each class. Details are held in the form of IDL descriptions of each method's interface (which are programming language-independent). This database is used during dynamic method invocation to find a suitable server object for a dynamic request by matching the IDL definition of each of the server's method prototypes against the IDL description in the request message.

6.11.1.5 Static Method Invocation

Static invocation is used when the client object wants to send a request message to a specific design-time-known object. The mechanism of static invocation is illustrated in Figure 6.42.

Figure 6.42 illustrates how static invocation takes place. The sequence of steps shown in the figure begins at the point where a client identifies a particular implementation object that it wishes to send a method invocation request to. The steps are as follows:

1. The client object sends the request message.
2. The message is passed to the client stub (associated with the client object's application) that represents the server object (an application has one such stub for each remote object that it may access).
3. The message is converted into IDL representation.
4. The IDL code is passed via the client-side ORB (for the client platform) to the network. The client-side ORB consists of local services that the application may use (e.g., it may convert an object reference to a string for type-safe transmission over the network).

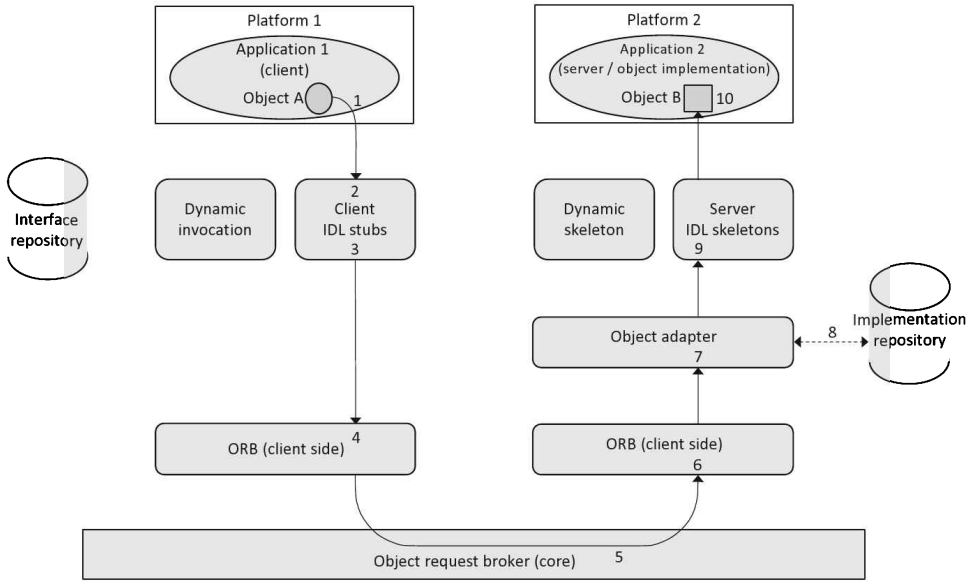


FIGURE 6.42

The mechanism of static method invocation.

5. The message is passed over the network through the ORB core. CORBA enforces strict syntax and the mode of transport for messages passed over the network to ensure interoperability between CORBA implementations. The required syntax is the General Inter-ORB Protocol (GIOP) and the mode of transport is the Internet Inter-ORB Protocol (IIOP), which operates over TCP.
6. Once the message arrives at the server object's platform, it is picked up by the client-side ORB (which may perform local services such as object reference format conversion) and passed to the object adapter.
7. The object adapter provides the runtime environment for instantiating server objects and passing requests to them.
8. The object adapter searches the implementation repository to find an already running instance of the required server object or, otherwise, instantiates and registers the newly instantiated server object with the repository.
9. The object adapter passes the message to the appropriate server skeleton, which translates the IDL message into the message format of the specific server object.
10. The server object receives the message and acts on it.

6.11.1.6 Dynamic Method Invocation

Dynamic method invocation is used when the client object requests a service (by description) but does not know the specific object ID or class of object to satisfy the request. This supports runtime-configured applications where the relationships between components are not determined at design time, that is, components are not tightly coupled (see the discussion on component coupling in Chapter 5).

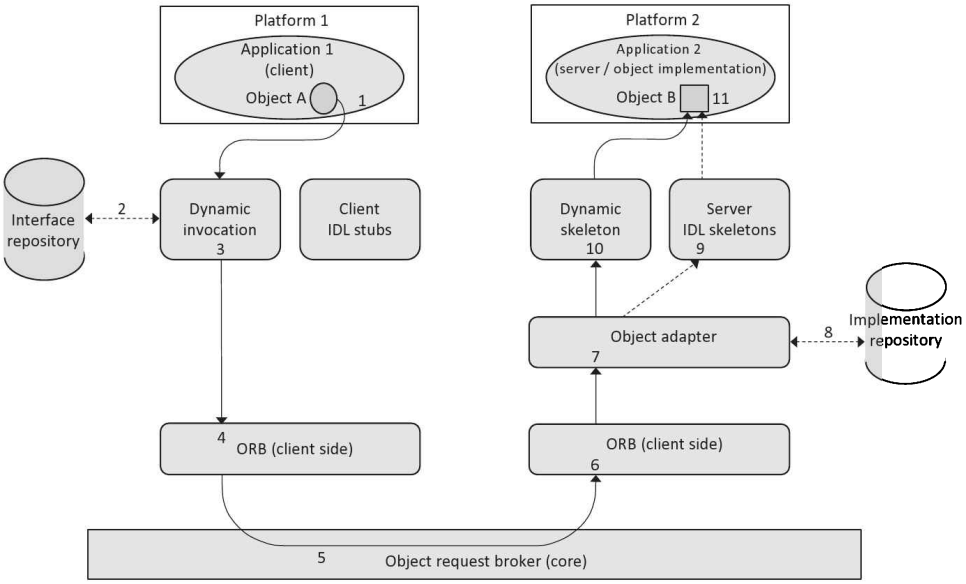


FIGURE 6.43

The mechanism of dynamic method invocation.

Figure 6.43 illustrates how dynamic method invocation takes place. The sequence of steps shown in the figure begins at the point where a client identifies a particular implementation object that it wishes to send a method invocation request to. The steps are as follows:

1. The client object sends the message to the dynamic method invocation utility.
2. The dynamic method invocation utility accesses an interface repository (a runtime database) that contains IDL descriptions of the methods associated with various objects. The dynamic method invocation utility identifies one or more objects that have methods that could satisfy the request.
3. The request is converted to IDL code and routed to those objects.
4. The IDL code is passed via the client-side ORB (for the client platform) to the network.
5. The message is passed over the network through the ORB core, using the GIOP and IIOP as with the static invocation mechanism.
6. The message is received by the client-side ORB (this time on the server-object's platform).
7. The message is passed via the object adapter to the appropriate object (which may need to be instantiated).
8. The object adapter searches the implementation repository to find an already running instance of the required server object or, otherwise, instantiates and registers the newly instantiated server object with the repository.
9. If the appropriate object on the server has an IDL skeleton, the message is passed via the server skeleton and thereby translated from IDL code into the message format of the target (server) object.
10. In the more complicated case, when the server object does not have an appropriate skeleton, the dynamic skeleton utility dynamically creates a skeleton for the server object and translates the IDL code into the message format of the target (server) object.
11. The server object receives the message and acts on it.

6.11.1.7 OMG Interface Definition Language (IDL)

The use of an IDL in middleware such as CORBA is a very powerful concept for application interoperability and in particular solves the problem of interoperability between objects written in different programming languages (and thus achieves implementation transparency). CORBA uses the OMG IDL, which is a specific instance of an IDL with support for several target languages including C, C++, and Java.

The OMG IDL is used to describe the interfaces to objects so that method calls can be made to them. IDL descriptions completely and unambiguously define all aspects of component call interfaces, including the parameters, their types, and the direction in which each parameter is passed (into the method or returned from the method). Using these interface definitions, a client component can make a method call on a remote object without knowing the language the remote object is written in and also without needing to know how the functionality is implemented; the IDL description does not include the internal behavior of the method.

The IDL interface definitions are needed during the automatic generation of the code (see below) in order that stubs and skeletons can automatically be generated and compiled into the application code. This removes the need for application developers to be concerned with communication and interoperability aspects and instead focus only on the business logic of components, as if all calls were between local objects.

See also the further discussion on IDL in a later section.

6.11.1.8 CORBA Application Development

Middleware such as CORBA provides design-time transparency to application developers and runtime transparency to applications and objects. The use of CORBA simplifies the application developer's role in building distributed applications because it facilitates interoperability between components and also provides mechanisms to take care of the networking and connectivity aspects.

The application development approach encouraged that developers write their code as if all components were to run locally and therefore not require any networking support. The developer must still take care of the separation of concerns and distribute the application logic across the components appropriately. Where there are static design-time-known relationships between components, these are built-in (this implies the use of the static invocation mechanism at runtime). So if, for example, a particular component *X* is known to need to contact component Y to perform some specific function, then this relationship will be built into the application logic, in the same way as static relationships between objects are embedded in nondistributed applications.

The application developer must however perform an additional step, which is to describe the interfaces between the components using OMG IDL.

6.11.1.9 Automatic Code Generation in CORBA

The user-provided IDL code specifies the interfaces of objects and thus defines how their methods are invoked. Based on the IDL definitions, the IDL compiler generates the client stubs and server implementation skeletons.

The stub and skeleton provide an object-specific interface between the object and the ORB, as shown in Figure 6.40. Significantly, this removes the need for the application developer to write any communication code, and because of transparencies provided by the various CORBA components and mechanisms, the application developer does not need to be concerned with any aspects of distribution

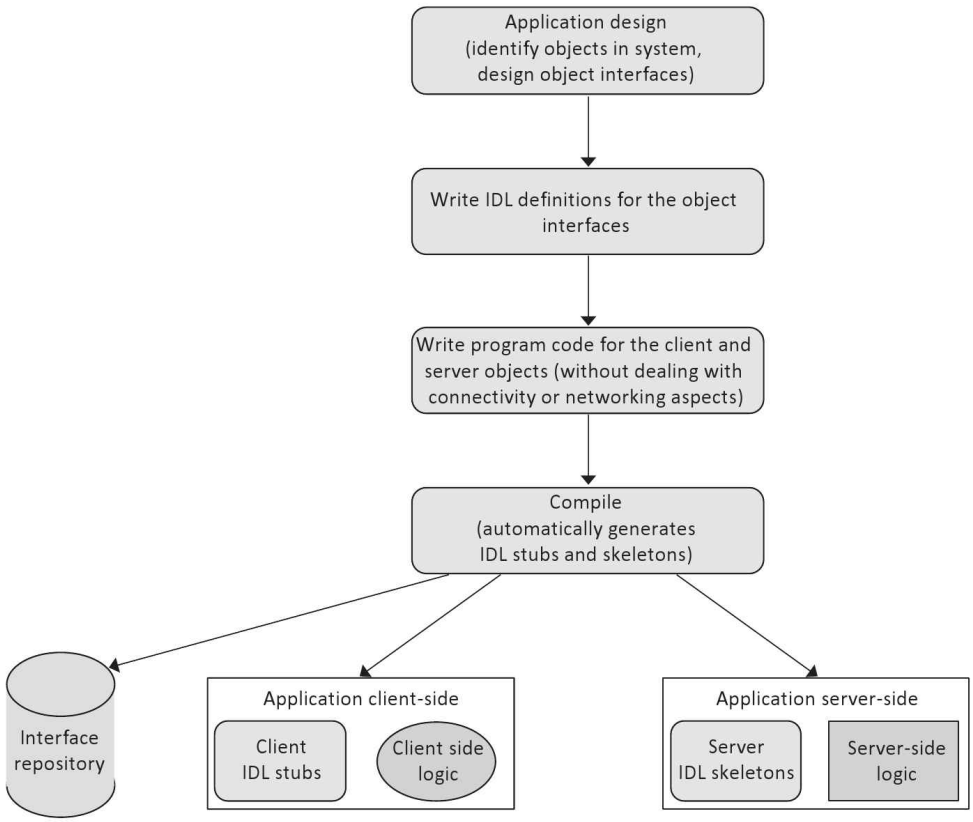


FIGURE 6.44

The sequence of steps to develop a CORBA application.

such as the relative locality or remoteness of objects and network addresses. The stubs and skeletons automatically include the necessary links between the objects and the CORBA mechanisms that deal with the actual network communication automatically.

The server implementation skeleton is used by the ORB and object adapter to make calls to method implementations. The IDL definitions provide the object adapter with the required format of each specific method call.

Figure 6.44 illustrates the basic sequence of steps in the development of CORBA applications. The developer's role is simplified because the complexity of the distribution, networking, and connectivity is handled automatically by CORBA's mechanisms that are driven by the IDL.

6.11.2 INTERFACE DEFINITION LANGUAGE (IDL)

The IDL approach separates the interface part of program code from the implementation part. This is very significant for distributed systems because these are two types of concern that should ideally be treated in different ways. Application developers need to focus on the business logic aspects of application behavior (i.e., the implementation). The separation of interface from implementation facilitates automation of interface-related aspects such as connectivity and communication functionality, through middleware services.

There are various IDLs, used in specific middlewares, but they tend to be similar in terms of the role they perform, the ways in which they are used, and their syntax. CORBA uses the OMG IDL, for example, while Microsoft defined the Microsoft IDL (MIDL) primarily for use with its other technologies such as OLE, COM, and DCOM.

An IDL is designed to be programming language-independent, that is, it defines the interface of a component only in terms of the names of methods and the parameter types that are passed in and out of the method, regardless of whether the method is implemented in C++, C#, Java, or some other language. An IDL explicitly identifies the type and direction of each parameter. The type values supported are the same in general as most languages and include int, float, double, bool, and string. The direction is defined explicitly (as being an input to the method (in), an output from the method (out) or a parameter that is passed in and modified, and the new value output (in, out)) instead of implicit representation in which the position of the parameter in addition to the use of special symbols (such as * and &) indicates the way the parameters are used. An example of an (in, out) parameter in conventional C usage is a value passed into a method by reference, whereby the called method may modify the variable (this is the same instance of the variable that is visible to the calling method, so the change in value will be seen when the call returns). In addition to the basic description of parameters, IDL can provide additional constraints (such as maximum and minimum values). Figure 6.45 puts the use of IDL into context with a simple example.

Figure 6.45 provides a very simple example of the OMG IDL syntax. Part A shows part of the C++ header file for a simple class called simple MathUtils. There are two simple member function declarations shown. Part B of the figure shows the equivalent OMG IDL representation. Notice that the IDL representation is quite similar to the C++ header file format; this is not surprising when you consider that the header file is essentially an interface specification and does not contain implementation detail. In the particular example shown (for C++), the IDL uses the term interface to replace class and places the parameter direction ahead of each listed parameter, to avoid any language-specific implied meaning. IDL does use implication in regard to the parameter to the left of the method name, which is implied to be an out parameter by its position (as with many high-level languages).

Part of a C++ class header showing two function prototypes

```
class SimpleMathUtils
{
...
int Add2(int iFirst, int iSecond);
float Add3f(float fFirst, float fSecond, float fThird);
...
}
```

(a)

The equivalent representation in OMG IDL

```
interface SimpleMathUtils
{
...
int Add2(in int iFirst, in int iSecond);
float Add3f(in float fFirst, in float fSecond, in float fThird);
...
}
```

(b)

FIGURE 6.45

Comparison of programming language-specific interface definitions and IDL.

The implementation of the application logic is contained within the components themselves and is not exposed at its interfaces; therefore, IDL does not need to represent the implementation. So for the example shown in Figure 6.45, the IDL representation does not show how the addition methods work. This detail is not required by the calling component, which only needs the information shown in the interface definition in order to invoke the methods; for this reason, IDL does not have language constructs to represent the implementation.

The most significant difference between the two representations of the same interface in Figure 6.45 is that the C++ specific one is only understood by a C++ compiler, and therefore, only another component written in C++ could call the methods. The IDL representation can be used with any of the supported languages (which are many) so the fact that the simple MathUtils server object has been developed in C++ does not place any restrictions on the language used to develop a client object that calls its methods.

6.11.3 EXTENSIBLE MARKUP LANGUAGE

Extensible Markup Language (XML) is a standard, platform-independent markup language that defines format rules for encoding data. A main strength is that it provides an unambiguous way to represent structured data for use within applications, in particular as a format for storage and for communication in messages.

XML's portability across different platforms makes it ideal for representing data in distributed systems, overcoming heterogeneity. The XML is also extensible, which enables application-specific and application domain-specific variations of the basic language to be created, a classic example of which is the Chemical Markup Language (CML); this is used to describe complex molecular structures in a standardized and unambiguous document format. XML's characteristics have led to its popular use in many applications, as well as being the data representation method of choice in other protocols such as SOAP and Web services. A simple example of the use of XML to encode structured data is provided in Figure 6.46.

```
<?xml version="1.0" encoding="UTF-8"?>
<CUSTOMER_LIST>
  <CUSTOMER>
    <CUST_ID> 000129365</CUST_ID>
    <CUST_NAME>John Jones</CUST_NAME>
    <CUST_ADDR>Victoria British_Columbia Canada</CUST_ADDR>
    <CUST_DOB>10_February_1976</CUST_DOB>
  </CUSTOMER>
  <CUSTOMER>
    <CUST_ID> 000031348</CUST_ID>
    <CUST_NAME>Mary Smith</CUST_NAME>
    <CUST_ADDR>Leeds Yorkshire England</CUST_ADDR>
    <CUST_DOB>22_May_1954</CUST_DOB>
  </CUSTOMER>
  <CUSTOMER>
    <CUST_ID> 000245170</CUST_ID>
    <CUST_NAME>Pierre Vert</CUST_NAME>
    <CUST_ADDR>Paris France</CUST_ADDR>
    <CUST_DOB>17_October_1982</CUST_DOB>
  </CUSTOMER>
</CUSTOMER_LIST>
```

FIGURE 6.46

XML encoding in a simple application example.

Figure 6.46 illustrates how XML provides a simple way in which to encode complex data structures in an unambiguous way. The data representation format not only is easy to parse within a computer program but also is human-readable, which contributes to usability significantly. Compare, for example, with simpler formats that could be considered alternatives, such as a comma-separated file in which each data field is separated by a delimiter such as a comma and the position in the list denotes the meaning of each field; this is illustrated in Figure 6.47.

Figure 6.47 presents the same data as encoded in Figure 6.46, but in a simple comma-separated list format. In this example, the field delimiter is a comma and the row delimiter is a semicolon. The comma-separated list format serves to illustrate the relative benefits of XML. The comma-separated list format is more efficient, but XML has several advantages: (1) It preserves the structure of the data. (2) It names each field explicitly, which makes the format more readable. (3) It supports repetition of several data items of the same type within a single data record; for example, consider extending both the XML and comma-separated representations of the customer data to encode customers' phone numbers. The XML format can easily deal with situations where a customer has zero, one, or more phone numbers, because each entry is explicitly labeled. However, the comma-separated format uses position to represent meaning so repeating or omitted fields are not so simple to encode.

6.11.4 JAVASCRIPT OBJECT NOTATION (JSON)

JSON is an efficient data-interchange format based on a subset of JavaScript and is both human-readable and straightforward for programs to create and parse. JSON is suitable for use in distributed systems because it uses a programming language-independent textual representation of data and thus provides implementation transparency in systems where multiple programming languages are used. JSON is a popular format for data interchange in Web applications.

A JSON script is organized in two levels. The outer level comprises a list of objects. Within this, each object is represented as a set of records, each expressed as an unordered list of name-value pairs delimited by commas. JSON is generally more concise than XML, a compromise between the simpler (raw) formats such as a comma-separated list and the highly structured XML. An application example of the JSON format is provided in Figure 6.48, using the same customer application as in Figures 6.46 and 6.47.

```
*CUSTOMER_LIST;
000129365,John Jones,Victoria British_Columbia Canada,10_February_1976;
000031348,Mary Smith,Leeds Yorkshire England,22_May_1954;
000245170,Pierre Vert,Paris France,17_October_1982;
```

FIGURE 6.47

Comma-separated list format—for comparison with XML.

```
JSON example
{" CUSTOMER_LIST ":[
  {"CUST_ID":"000129365", "CUST_NAME":"John Jones", "CUST_ADDR":"Victoria British_Columbia Canada", "CUST_DOB":"10_February_1976"},
  {"CUST_ID":"000031348", "CUST_NAME":"Mary Smith", "CUST_ADDR":"Leeds Yorkshire England", "CUST_DOB":"22_May_1954"},
  {"CUST_ID":"000245170", "CUST_NAME":"Pierre Vert", "CUST_ADDR":"Paris France", "CUST_DOB":"17_October_1982"}
]}
```

FIGURE 6.48

JSON encoding in a simple application example.

Figure 6.48 shows a JSON script that defines a single CUSTOMER_LIST object, which contains an array of 3 customer records. The example illustrates how JSON shares some efficiency characteristics with the comma-separated list format while also retaining the explicit labeling of XML, making it flexible with respect to the representation of complex data with repeating fields or omitted field values and enhancing human readability. The JSON usage of arrays and lists more naturally matches the data model used in most programming languages than XML does, and therefore, it is more efficient to parse.

6.11.5 WEB SERVICES AND REST

A Web service is a communication interface that supports interoperability in distributed systems using standard Internet protocols used widely in the World Wide Web (Web) and as such is naturally platform-independent. The Web Services Description Language (WSDL) is used to describe the functionality of a Web service (WSDL is a type of IDL based on XML).

Clients communicate with the Web service using the Simple Object Access Protocol (SOAP), which is also a platform-independent standard (see later text). The SOAP messages themselves are encoded using XML and transmitted using the http, two further standards. The service is addressed using its URL, which describes the specific service and also represents the address of the host computer.

A main class of Web services is those that are REST-compliant. These Web services use a fixed set of operations—PUT (create), GET (read), POST (update), and DELETE—to manipulate XML representations of Web resources (see "Representational State Transfer (REST)" below). There are also arbitrary Web services, which are not constrained by the rules of REST and can thus expose an arbitrary set of operations. A simple example of a Web service application is shown in Figure 6.49.

Figure 6.49 illustrates a simple Web service following the same Customer Details application scenario as the previous sections. The example shows a single stateless Web service with three GET methods {GetCustomerName, GetCustomerAddress, GetCustomerDOB}. The method requests are parameterized by the string value CustomerID. In other words, the Web service does not need to store any state concerning the client or the client's request; all the necessary information is self-contained in the request message. This is an important aspect that contributes to the scalability of Web services.

6.11.5.1 Representational State Transfer (REST)

REST is a set of guidelines that are designed to ensure high quality in applications such as Web services, in terms of simplicity, performance, and scalability. REST-compliant (or RESTful) Web services must have a client-server architecture and use a stateless communication protocol such as http.

The design of RESTful applications should respect the following four design principles:

1. Resource identification through URI: Resources that are accessible via a RESTful Web service should be identified by URIs. URIs represent a global Web-compliant-related address space.
2. Uniform interface: A fixed set of four operations—PUT, GET, POST, and DELETE—are provided to create, read, update, and delete resources, respectively. This restriction ensures clean, uncluttered, and universally understood interfaces.
3. Self-descriptive messages: Resources need to be represented in various formats, depending on the way they are to be manipulated and how their content is to be accessed. This requires that the representation of a resource in a message is decoupled from the actual resource itself and that the request and response messages identify the resource itself and either which operation is to be performed or the resulting value of the resource after the operation, respectively.

```
<%@ WebService language="C" class="CustomerDetails" %>

using System;
using System.Web.Services;
using System.Xml.Serialization;

[WebService(Namespace="http://www.widgets.org/customerdetails/WebServices/")]
public class customerdetails : WebService
{
  [WebMethod]
  public String GetCustomerName (String CustomerID)
  {
    ...
    return CustomerName;
  }
  [WebMethod]
  public String GetCustomerAddress (String CustomerID)
  {
    ...
    return CustomerAddress;
  }
  [WebMethod]
  public String GetCustomerDOB (String CustomerID)
  {
    ...
    return CustomerDOB;
  }
}
```

FIGURE 6.49

A simple Web service application example.

4. Stateful interactions through hyperlinks: The Web service itself, and thus each server-side interaction with a resource, should be stateless. This requires that request messages must be self-contained (i.e., the request message must contain sufficient information to contextualize the request so that it can be satisfied by the service without the need for any additional server-side-stored state concerning the client or its specific request).

6.11.6 THE SIMPLE OBJECT ACCESS PROTOCOL (SOAP)

The SOAP facilitates connectivity in heterogeneous systems and is used to exchange structured information in Web services. It uses several standard Internet protocols to achieve platform and operating system-independent message transmission and message content representation. Message transmission is usually based on either hypertext transfer protocol (http) or Simple Mail Transfer Protocol (SMTP).

http and the XML are used to achieve information exchange in a format that is both universally recognized and unambiguous in interpretation. SOAP defines how an http header and an XML file should be encoded to create a request message and the corresponding response message so that components of distributed applications can communicate. Figure 6.50 illustrates the use of SOAP with a simple application example.

SOAP request (get customer name from ID)

```
POST /customerdetailsHTTP/1.0
Host: www.widgets.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length:nnn

< ?xml version="1.0"?>
< soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Bodyxmlns:m=" http://www.widgets.org/customerdetails ">
    <m:GetCustomerName>
      <m:CustomerID>000129365</m: CustomerID>
    </m: GetCustomerName>
  </soap:Body>
< /soap:Envelope>
```

(a)

SOAP response (return the requested customer name)

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

< ?xml version="1.0"?>
< soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Bodyxmlns:m=" http://www.widgets.org/customerdetails ">
    <m: GetCustomerNameResponse>
      <m: CustomerName>John Jones</m: CustomerName>
    </m: GetCustomerNameResponse>
  </soap:Body>
< /soap:Envelope>
```

(b)

FIGURE 6.50

SOAP encoding in a simple application example.

Figure 6.50 provides a simple application example to illustrate the way in which SOAP combines HTML and XML to define unambiguous request and response message formats. Part A of the figure shows the SOAP request to get the name of a customer, based on a provided customer ID value. This is an http POST request type. Within the message, XML coding format is used to represent the structure of the request message data. Part B of the figure shows the corresponding response message.

6.12 DETERMINISTIC AND NONDETERMINISTIC ASPECTS OF DISTRIBUTED SYSTEMS

Deterministic behavior essentially means predictable. If we know the initial conditions of a deterministic function, we can predict the outcome, whether it be a behavior or the result of a calculation.

Nondeterministic behavior arises in systems in which there are sufficient numbers of interacting components and sources of randomness that it is not possible to predict the future states precisely. Natural systems tend to be nondeterministic; examples include behavior in insect colonies, weather systems, and species population sizes. Human-oriented nondeterministic system examples include

economies and crowd behavior. These systems are sensitive to their starting conditions and the actual sequences of interactions that occur within the system and possibly even the timing of those interactions. For such systems, it is usually possible to predict an expected range of outcomes, with varying confidence levels, rather than a knowable particular outcome. Computer simulations of nondeterministic systems may have to be run many times in order to gain usable results, with each run having slightly different starting conditions and yielding one possible outcome. Patterns in the set of outcomes may be used to predict the most likely of many possible futures.

For example, a weather forecasting algorithm is very complex and uses a large data set as its starting point in order to forecast a weather sequence. It is actually a simulation, computing a sequence of future states based on current and recent conditions. There may be some tuning parameters that affect the algorithm's sensitivity to certain characteristics of the input data. As there are so many factors affecting the accuracy of the weather forecast and also the fact that we cannot capture all the possible data needed with equal accuracy across all samples (sensors and/or their placement may be imperfect), there is always an element of error in the result. If we run the simulation just once, we may get a very good forecast, but it may also be poor (because of the varying sensitivity to specific weather histories and varying extents of dynamism in the weather systems). So although we may have the best forecast we are going to get, we cannot be confident that that is the case. If we change the tuning parameters very slightly and run the simulation again, we may get a similar or distinctly different result. Running the simulation many times with slightly different settings will give us a large number of outcomes, but hopefully, they are clustered together such that if somehow averaged, they provide a good approximation of what the weather will actually do. This is an example of a nondeterministic computation that hopefully yields a valuable result.

Distributed computer systems are themselves complex systems with many interacting parts. Nondeterministic behavior arises due to a large number of unpredictable characteristics of systems. For example, network traffic levels are continuously fluctuating, and this leads to different queue lengths building up, which in turn affects the delay to packets and also the probability of packets being dropped because a queue is full. Even a single hardware component can contribute nondeterministic behavior; consider the operation of a hard disk. Disk seek time is dependent on the rotation delay, which depends on where the disk is (in an angular sense) at the point of starting the seek and also the relative distance between the current track and one the head has to move to. These variations in starting conditions affect every block read from the disk so it is more likely that disk read times will vary each time than they will be exactly the same. The runtime of even a simple process depends on the sequence of low-level scheduling decisions made by the operating system, which in turn depend on the set of other processes present and the behavior of those processes. Therefore, executing a single process multiple times can lead to different runtimes, arising from the combination of differences in scheduling decisions, disk access latency, and communication latency.

The message for designers and developers of distributed applications and systems is that even if your algorithms are entirely deterministic in their operation and even if your data are complete and accurate, the underlying computing systems and networks are not perfect and not perfectly predictable. A component failure can happen at any time, a network message can be corrupted, the load on a processing host computer can change suddenly leading to increased latency of service, and so on. Seeking to eradicate all unpredictable behavior is futile, and the assumption of determinism is dangerous. You cannot prevent certain types of faults, and you cannot in general predict them. Instead, focus your design effort on making applications robust and fault-tolerant.