Module 9

Distributed Software Engineering; Service-Oriented Software Engineering

Objectives:

1. Understand some of the basic advantages and disadvantages to developing and using distributed software systems.
2. Be introduced to several common patterns for distributed systems architectures.
3. Understand how the client/server model works.
4. Be introduced to the notion of providing software as a service.
5. Understand the basic notions of web services and service-oriented architectures, and know the primary web service standards that manage how web service components are created.
6. Understand how the service-oriented approach differs from the component-oriented approach.

Reading:

Sommerville, chapters 17 & 18.

Assignment:

No assignment for Module 9.

Quiz:

No quiz for Module 9.

Some Basic Pros and Cons of Distributed Systems

Large-scale software systems used to be confined to a single computer or large mainframe, and utilized only a single processor. The current trend is towards a **distributed architecture**, which usually implies the presence of multiple computers, but in some cases can refer to multiple processors on the same machine. What are some advantages and disadvantages of using a distributed architecture?

Some key advantages to using a distributed architecture are, as listed in the text:

1. Resource Sharing

   Most, if not all of the hardware and software of a distributed system can be shared by all the computers in the system. This can result in a significant cost savings over acquiring and maintaining an equivalent number of individual computers. In most cases, peripheral hardware such as printers and hard drives spend a lot of time sitting idle when attached to a non-distributed system. Connecting these devices to a distributed system can drastically reduce idle time, resulting in more productivity from those devices.

2. Openness

   By openness, we are referring to the fact that distributed systems can be designed to follow certain standards that allow consistent operation even though the specific hardware and software configurations for the computers in the system may differ.

3. Concurrency

   Since there are multiple processors available, multiple software processes can be running truly simultaneously, as opposed to the pseudo-concurrency achieved by scheduling of multiple processes on a single processor. Depending on the processes and the system architecture, it is possible to have processes running on different processors communicate with each other, if necessary.

4. Scalability

   Theoretically, a distributed system should be able to scale to any size of task by simply adding more resources (e.g., memory, disk space, processors). The actual scalability of a distributed system, however, will generally be limited by a bottleneck in one or more resources—most likely network bandwidth availability.

5. Fault Tolerance

   We haven't covered the notion of fault tolerance yet, but in a nutshell fault tolerance allows a system to continue normal operation in the event an error occurs. Just to clarify some terminology, a **fault** is a mistake in code (i.e., a bug), a physical hardware defect, or a "hiccup" (e.g., a momentary voltage change prevents a byte from being read or written); an **error** is an unexpected system state that occurs because of a fault; a **failure** is abnormal or unexpected behavior of a system that results from an error. In a distributed system, fault tolerance can be implemented by running multiple copies of the same software simultaneously, and/or maintaining multiple copies of the data being processed. The key here is the redundancy—it isn't likely that all the software copies will experience the same error at the same time, or that all the copies of the data will be corrupted at the same time.

So far, so good. Distributed systems certainly seem to offer some attractive advantages. But what are some disadvantages/issues of distributed systems? The primary issues are:

1. Increased Complexity

   Distributed systems tend to be far more complex than non-distributed systems. Special considerations need to be made when designing concurrency into software. Difficulties can arise when attempting to ensure openness, so that different hardware and software configurations can be supported. There is also the notion of **emergent properties**, which arise when interactions between the different components in a distributed system lead to behaviors that would not arise in a non-distributed system. In other words, emergent properties can cause the whole system to be apparently greater than the sum of its parts. Emergent properties have been well-known in biology for years, with ant colonies and bee colonies being prime examples. Individual drones are essentially powerless alone, but when they work in concert with the rest of the colony, a very complex society develops. In general, emergent properties can occur in any situation where the behavior of a system as a whole is influenced by dynamics that are created only by the parts of the whole behaving (apparently) independently.

2. Load

   Although a distributed system allows true concurrency, it can still be overwhelmed if too heavy a load is placed on it. The denial of service problems that periodically slow down or bring down portions of the internet are prime examples of this.

3. Non-compliance to Standards

   Ideally, the components of a distributed system should follow open standards to ensure the system is not dependent on specific hardware or software configurations. If open standards are not followed, problems can arise. One excellent example that illustrates this is the browser wars that peaked around a decade ago, and still cause some problems even today. Web applications that take advantage of browser capabilities that do not conform to open standards coerce users of those applications to adopt specific software configurations, which breaks the openness of a distributed system.

4. Security

   The computers in a distributed system may be in separate geographic locations, and managed by different groups of individuals. This makes it difficult to ensure uniform security policies are enforced consistently. Security in a distributed system must defend against several types of attacks, including interception of communications (e.g., packet sniffers), service interruptions (e.g., DoS attacks), hacking into a system to gain unauthorized access to resources and data, and changes made to services or data via hacking.

5. Quality of Service

   Users of a distributed system may understand that certain problems are to be expected from time to time, such as periodic network slowdowns, but they will still expect a certain amount of quality in the service provided. Ensuring that a minimum quality of service is provided is therefore essential, but represents a significant problem, particularly if the distributed system is a critical system.

6. Failure Management

   Despite the best efforts to ensure fail-free operation, sooner or later a failure of some type is bound to occur. Sophisticated systems based on a fault tolerant architecture can detect some errors and correct them before they lead to serious failures. In the event a failure cannot be managed, policies need to be in place to ensure the system is brought back online as soon as possible. In many cases this will mean bringing the system back online while the fault that gave rise to the failure is still present, with immediate efforts initiated to

find the fault and correct it before the failure happens again.

7. Transparency

   The notion of transparency means that users of a distributed system should have the experience of using an isolated computer. In other words, they should not be able to tell from a usage standpoint that they are using a distributed system, even though they may know beforehand that the system is distributed. Complete transparency in practically impossible to achieve, because network bottlenecks occur periodically, different computers in the system may behave differently from time to time, etc.

Middleware

Because the computers in a distributed system can have different hardware and software configurations, they may not be capable of talking directly to each other, yet computer-to-computer interaction is required for a distributed system to work. For that reason, special software known as middleware is developed to serve as a translator or liaison between different computers in the system. The term "middleware" is broad, encompassing single applications as well as entire libraries of components. Middleware can be classified into four major types:

1. Transactional Middleware

   Transactional middleware supports distributed synchronous transactions. An example of this type of middleware is IBM's CICS (Customer Information Control System). Transactional middleware should ideally support four properties that are collectively known as the ACID properties, an anagram for Atomic, Consistent, Isolated, and Durable. An atomic transaction is one that is either fully completed or not completed at all; no partially completed transactions are allowed. The system must always be in a consistent state regardless of the status of any given transaction. Isolated means that all transactions should be capable of being completed independently of other transactions. Durable transactions are capable of surviving system failures, provided they have been committed.

2. Message-Oriented Middleware

   Message-oriented middleware (MOM for short) is designed to support both synchronous and asynchronous communication between computers via messages, which are basically just strings of bytes that have special meaning to the systems using the appropriate middleware. Examples of MOM include IBM's MQSeries and Java Message Queue.

3. Procedural Middleware

   This type of middleware is represented by the remote procedure call (RPC) software found with most operating systems. Remote procedure calls allow programs located on one computer to call subroutines or procedures on other (remote) computers. This is common in client/server systems where the client remotely calls procedures located on the server.

4. Object-Oriented Middleware

   Object-oriented middleware, for the most part, encapsulates remote procedures into objects using object-oriented design techniques, and allows these objects to be referenced across the computers in a distributed system. Some examples of object-oriented middleware are CORBA (Common Object Request Broker Architecture), Microsoft's COM (Component Object Model), and JavaBeans.

Distributed System Architectures

Sommerville describes the most commonly used architectures for designing distributed systems:

1. Master-slave

   This architecture is used in real-time systems when precise response times between components are crucial. In this architecture there is a single, master process that coordinates and manages one or more slave processes. The slave processes are usually relegated to pe forming relatively specific tasks such as acquiring data from a sensor or performing a processor-intensive calculation.

2. Two-tier Client/Server

   The two-tier architecture is exemplified by the well-known client/server model. In this architecture, one or more client computers remotely access a single server. A layered model is used, containing 4 layers:

   a. *Presentation layer* — manages how information is presented to the user and handles user interactions

   b. *Data Management Layer* — handles data as it is passed between the client and the server, possibly performing validation checks and other tasks

   c. *Application Processing Layer* — contains the application and/or business logic that encapsulate the system's functionality as detailed in the requirements specification

   d. *Database Layer* — responsible for storing data and providing services for data retrieval, data entry, and data modification

   There are two flavors of client available in a client/server architecture: *thin-client* and *fat-client*, that differ in how much of the application layer is present. A thin client has none of the application layer present, while a fat client contains some, and possibly all of the application logic.

3. Multitier Client/Server

   The multitier architecture extends the traditional two-tier client/server architecture by splitting up the functionality of the various layers into additional, separate servers, or tiers. Multitiered systems are more complex, but they are inherently more scalable, and they do a better job of modularizing the overall architecture.

4. Distributed Component Architecture

   The layered client/server model can pose problems for developers in that decisions must be made during design as to which layer should contain a particular piece of functionality. This can complicate future extension of the system, and may impact the scalability of the system. This is where the service-oriented approach comes in. By designing the system as a set of independent, but interacting services, rather than pigeon-holing functionality into discrete layers, the flexibility of the system can be maximized.

5. Peer-to-Peer

   All client/server-based architectures clearly distinguish between the client computers and the server computers. An obvious problem that can occur with this architecture is that if the server becomes burdened with requests, the entire system may be slowed down. Additionally, if the server is hacked, the system is completely shut down. These problems can be mitigated by decentralizing the system such that there is no distinction between the clients and the server. The resulting architecture is known as peer-to-peer, or P2P for short. In a P2P system, any computer can act as the client or the server as needed, depending on the

available resources at a given time. The prime example of a P2P system is represented by the many of the popular file sharing networks commonly used by many people to share all types of media content (some of which violates copyright and licensing laws). The VoIP application, Skype, also uses a P2P architecture.

Software as a Service

In a client/server system where a thin client is used, most of the processing is done on the server, which can result in the server being easily overburdened. The fat client approach relieves the pressure on the server by moving more processing to the client, but this creates a problem with management of the application logic. If most of the logic is on the client, and the logic and/or functionality change, the client software must be updated accordingly. Ensuring that all the clients are updated with the latest version of the software can be very difficult, and can be a source of consternation on the part of end users. One of the big advantages, therefore, to using thin clients is that developers have more control over the functionality, by virtue of the fact that most of the functionality resides on a single computer. If the functionality changes, or a bug needs to be fixed, the necessary modifications only need to be done in one place.

In an attempt to have the best of both worlds, providing software as a service entails hosting the application logic on the server while allowing client computers to use a standard web browser as the user interface. End users thus run the software application remotely. This alleviates to an extent the problem of having to ensure all clients are running the latest client software. The web browser will need to be kept updated, but most people don't have too much of an issue with this since it is something they are likely to do anyway, and they may even have their browsers set to automatically download and install updates as they become available. Depending on the software application being used, clients may need to have certain browser plugins installed, and there can be problems with plugin conflicts, failure of end users to update the plugins, and so on.

Providing software as a service still has issues, though. If too many clients try to access a remote application concurrently, the server can become overwhelmed just as it can in a thin-client model. Performance can be slowed considerably, or even brought to a complete halt. If the hosted application is user-configurable, a way must be provided to retain user-specific changes in the default configuration. This could be done by storing client profiles on the server, for situations where a more or less static set of clients use the application. If the client population can fluctuate, it is probably best to store user-specific configuration data in a file on the client. Remote access to a client's file system through a web browser is usually severely limited, but information can be stored in a designated "temp" directory, or another directory as determined by the browser. If a user-defined configuration profile is lost, access to the application is still possible through the default configuration profile. Security can also be an issue. If sensitive data must be passed to the server for processing by the hosted application, a vulnerability is introduced whereby the data could be viewed by an intruder, or saved on the server or another computer.

Service-Oriented Architectures

Although providing software as a service and service-oriented architecture (SOA) both use the term "service", they are distinctly different in nature. While providing software as a service involves hosting a complete software application on a server and allowing clients to use the application through a web browser, a service-oriented architecture is a software development concept where distributed software is built from reusable components that function as services. These components are designed to be platform and programming language independent, and they are highly standardized. The components provide their functionality as a service to clients that request it. Clients, or service requestors, locate a service provider, bind themselves to a service, and communicate with it using standardized protocols. According to Sommerville, there are three main standards for web-based service-oriented architectures, all of which are XML-based:

1. SOAP (Simple Object Access Protocol), which standardizes communication between services

2. WSDL (Web Service Definition Language), which standardizes service interfaces; also, it enforces specification of what a web service does, where it can be found, and how it communicates with other services (operation names, parameters, and data types)

3. WS-BPEL (Web Services Business Process Execution Language), which is an executable language that can be used to define business workflow processes.

One key feature of a SOA is that the services are loosely coupled. As a system built from web-based service components executes, it will never know exactly what services are available at any given time. The system sends requests to a service discovery service, which locates appropriate services to satisfy the request, and binding occurs between the system and the selected service. The binding may be broken and re-established many times, and different services may be bound in different situations. Only a loosely coupled architecture could efficiently and effectively handle such dynamics. The loose coupling of service components also inherently promotes their use as reusable components. As we discussed in the previous module, reusable service components differ from standard CBSE components in that service components are designed to operate consistently, and independently of other components, regardless of the execution environment.

The key differences between the service-oriented approach and the component-oriented approach are discussed by Sommerville in the text, but I've reproduced that here, for convenience:

1. Services can be offered by any service provider inside or outside of an organisation. Assuming these conform to certain standards (discussed below), organisations can create applications by integrating services from a range of providers. For example, a manufacturing company can link directly to services provided by its suppliers.

2. The service provider makes information about the service public so that any authorised user can use the service. The service provider and the service user do not need to negotiate about what the service does before it can be incorporated in an application program.

3. Applications can delay the binding of services until they are deployed or until execution. Therefore, an application using a stock price service (say) could dynamically change service providers while the system was executing.

4. Opportunistic construction of new services is possible. A service provider may recognise new services that can be created by linking existing services in innovative ways.

5. Service users can pay for services according to their use rather than their provision. Therefore, instead of buying an expensive component that is rarely used, the application writer can use an external service that will be paid for only when required.

6. Applications can be made smaller (which is important if they are to be embedded in other devices) because they can implement exception handling as external services.

7. Applications can be reactive and adapt their operation according to their environment by binding to different services as their environment changes.