Chapter 18 Notes

Distributed System = a system involving several computers, in contrast with centralized systems where all of the system component execute on a single computer. Is a collection of independent computers that appears to the user as a single coherent system.

Advantages of using a Distributed approach to systems Development:

* Resource Sharing
  + Allows the sharing of hardware and software resources (disks, printers, files, and compilers) that are associated with computers on a network
* Openness
  + Are normally open system, which means that they are designed around standard protocols that allow equipment and SW from different vendors to be combined
* Concurrency
  + Several processes may operate at the same time on separate computer on the network and they may communicate with each other
* Scalability
  + Are scalable in that the capabilities of the system can be increased by adding new resources to cope with new demands on the system.
* Fault Tolerance
  + The availability of several computers and the potential for replicating information means that distributed system can be tolerant of some hardware and SW failures.

Disadvantages

* Distributed System are inherently more complex than centralized systems
* Harder to understand the emergent properties of these systems because of the complexity of the interactions between the system component and the system infrastructure
* These systems are unpredictable in their response

Distributed System Issues (Section 18.1)

* Important design issues that must be considered in Distributed Systems Engineering:
  + Transparency
    - To what extent should the system appear to the users as a single system? When is it useful for the users to understand that the system is distributed?
    - This is pretty much impossible to achieve completely
      * Network delays are unavoidable
  + Openness
    - Should a system be designed using standard protocols that support interoperability or should more specialized protocols be used that restrict freedom of the designer?
    - Openness is achieved at the network level through the use of networking protocols but openness in components is still not universal.
  + Scalability
    - How can the system be constructed so that it is scalable (ability to deliver high QoS as demands on the system increase)?
    - 3 Dimensions of Scalability
      * Size
        + It should be possible to add more resources to a system to cope with increasing numbers of users
      * Distribution
        + It should be possible to geographically disperse the components of a system without degrading performance
      * Manageability
        + It should be possible to manage a system as it increases in size, even if parts of the system are located in independent organizations
    - Scaling Up = replacing resources in the system with more powerful resources
    - Scaling Out = adding additional resources to the system (e.g., adding another server)
      * This is often more cost effective than scaling up but usually means that the system has to be designed so that concurrent processing is possible
  + Security
    - How can usable security policies be defined and implemented that apply across a set of independently manages systems?
    - Distributed systems have many more ways in which it can be attacked. Types of attacks are:
      * Interception
        + Communications between parts of the system are intercepted by the attacker so that there is a loss of confidentiality
      * Interruption
        + System services are attacked and cannot be delivered as expected. Example: DoS attacks
      * Modification
        + Data or services in the system are changed by an attacker
      * Fabrication
        + Attacker generates information that should not exist and then uses this to gain privileges

Different organization may own different parts of the system

* + - Difficulties
      * Difficulty in establishing a security policy that can be reliably applied to all of the components in a system
  + Quality of Service
    - How should the quality of service that is delivered to users be specified and how should the system be implemented to deliver an acceptable QoS?
    - Ideally, QoS requirements should be specified in advance and the system designed and configured to deliver that QoS. This is not always practicable for 2 reasons
      * It may not be cost effective to design and configure the system to deliver a high QoS under peak load. This could involve making resources available that are unused for much of the time.
      * The QoS parameters may be mutually contradictory.
        + Ex. Increased reliability may mean reduced throughput, as checking procedures are introduced to ensure that all system inputs are valid
    - This is critical if the system is delivering time-critical data like sound or video streams
  + Failure Management
    - How can system failures be detected, contained, and repaired?
    - System failures are inevitable so the system has to be designed to be resilient to these failures
    - Should apply fault tolerance techniques
* **Models of Interaction (Section 18.1.1)**
  + 2 Fundamental Types of Interaction in Distributed Systems
    - Procedural Interaction
      * One computer calling on a known service offered by some other computer and usually waiting for that service to be delivered
    - Message-Based Interaction
      * Involves the “sending” computer defining information about what is required in a message, which is then sent to another computer. Message usually transmit more information in a single interaction than a procedure call to another machine
* **Middleware (Section 18.1.2)**
  + Components in a distributed system may be implemented in different programming languages and may execute on completely different types of processor (thus, models of data, information representation, and protocols for communication may all be different). A distributes system therefore requires SW that can manage these diverse parts, and ensure that they can communicate and exchange data -> Middleware
  + Middleware sits in the middle between the distributed components of the system.
  + Middleware is normally bought off the shelf rather than written specially by application developers.
  + Middleware provides 2 types of support:
    - Interaction support
      * Middleware coordinates interactions between different components in the system. The middleware provides location transparency in that it isn’t necessary for components to know the physical locations of other components. It may also support parameter conversion if different programming languages are used to implement components, event detection, and communication, etc.
    - Provision of Common Services
      * Middleware provides reusable implementations of services that may be required by several components in the distributed system. By using these common services, components can easily interoperate and provide user services in a consistent way.

**Client-Server Computing (Section 18.2)**

* In a Client-Server system, the user interacts with a program running on their local computer (e.g. a web browser) which then interacts with another program running on a remote computer (e.g. web server). The remote computer provides services.
* Client and Server can be on the same machine
* Client-Server systems depend on there being a clear separation between the presentation of information and the computations that create and process that information. You should design the architecture of distributed client-server systems so that they are structured in several logical layers, with clear interfaces between them:
  + Presentation Layer
    - Concerned with presenting information to the user and managing all user interaction
  + Data Management Layer
    - Manages the data that is passed to and from the client. They layer may implement checks on the data, generate, web pages, etc.
  + Application Processing Layer
    - Concerned with implementing the logic of the application and so providing the required functionality to end user
  + Database
    - Layer that stores the data and provides transaction management services, etc.

**Architectural Patterns for Distributed Systems (Section 18.3)**

* When designing a distributed application, you should choose an architectural style that supports the critical non-functional requirements of your system. The 5 architectural styles are:
  + Master-Slave Architecture
    - Used in real-time systems in which guaranteed interaction response times are required
  + Two-Tier Client-Server Architecture
    - Use for simple client-server systems, and in situations where it is important to centralize the system for security reasons. In such cases, communication between the client and server is normally encrypted.
  + Multitier Client-Server Architecture
    - Use when there is a high volume of transactions to be processed by the server
  + Distributed Component Architecture
    - Used when resources from different systems and databases need to be combined, or as an implementation model for multi-tier client-server systems
  + Peer-to-peer Architecture
    - Used when clients exchange locally stored information and the role of the server is to introduce clients to each other. It may also be used hen a large number of independent computations may have to be made.
* **Master-Slave Architecture (Section 18.3.1)**
  + Commonly used in real-time systems where there may be separate processors associated with data acquisition from the system’s environment, data processing, and computation and actuator movement.
  + Master process is usually responsible for computation, coordination, and communications and it controls the Slave processes.
  + Slave processes are dedicated to specific actions, such as the acquisition of data from an array of sensors.
  + You can use this model in situations where you can predict the distributed processing that is required, and hwere processing can be easily localized to salve processors.
* **Two-Tier Client-Server Architecture (Section 18.3.2)**
  + Is the simplest form of Client-Server Architecture
  + The system is implemented as a single logical server and an indefinite number of clients that use that server.
  + 2 models
    - Thin-Client Model
      * Where the present layer is implemented on the client and all other layers (Data management, application processing, and database) are implemented on a server. The client software may be a specially written program on the client to handle presentation. More often, however, a web browser on the client computer is used for presentation of data
      * Pros
        + It is simple to manage the clients
      * Cons
        + May be difficult and expensive to install new software on all clients if there are many of them
        + May place heavy processing load on both the server and the network
    - Fat-Client Model
      * Some or all of the application processing is carried out on the client. Data management and database function are implemented on the server.
      * Pros
        + Distributes computation over more since the client performs some or all application processing
      * Cons
        + Requires additional system management to deploy and maintain the software of the client computer
* **Multi-Tier Client Server Architectures (Section 18.3.3)**
  + With 2-Tier client server architecture, presentation, application processing, data management, and database must be mapped onto 2 computer systems (the client and the server). This may lead to problems with scalability and performance (for thin-client model) and problems of system management (for the fat-client model).
  + Here, the different layers of the system namely presentation, data management, application processing, and database, are separate processes that may execute on different processors.
  + Pros
    - More scalable than 2-Tier Client-Server
    - Easy to update the application processing of the system since it is centrally located.
* **Distributed Component Architectures (Section 18.3.4)**
  + Organizing processing in to layers enables each layer of a system to be implemented as a separate logical server. But, it limits that flexibility of system designers in that they have to decide what services should be included in each layer.
  + This is a more general approach to distributed system design, were these services are NOT allocated to layers in the system. Each service, or group of related serices, is implemented using a separate component. In Distributed Component Architecutre, the system is organized as a set of interacting components or objects. These components provide an interface to a set of services that they provide. Other components class on these services through middleware, using remote procedure or method calls.
  + Benefits
    - Allows the system designer to delay decisions on where and how services should be provided. Service-providing components may execute on any node of the network. There is no need to decide in advance whether a service is part of a data management layer, application layer, etc.
    - Is a very open system architecture that allows new resources to be added as required.
    - System is flexible and scalable. New components or replicated components can be added as the load on the system increases, without major disrupting other parts of the system
    - Is possible to reconfigure the system dynamically with components migrating across the network as required. This may be important where there are fluctuating patterns of demand on services.
  + Cons
    - They are more complex to design than Client-Server systems. Multi-layer client-server systems appear to be fairly intuitive way to think about systems. By contrast, distributed component architectures are more difficult for people to visualize and understand.
    - Standardized middleware for this model has never been accepted by the community.
  + In this model, you can think about how to provide application functionality solely in terms of services and combinations of services. You then work out to provide these services using a set of distributed components.
  + This model should be used for high-throughput systems in which large numbers of transactions have to be processed quickly.
* Peer-to-Peer Architecture (Section 18.3.5)
  + Client-Server usually leads to uneven distribution of load on the systems, where servers do most of the work.
  + P2P systems are almost always decentralized systems in which computation may be carried out by any node on the network. No distinctions are made between clients and servers.
  + The standards and protocols that enable communications across the nodes are embedded in the application itself and each node must run a copy of that application
  + When to use P2P
    - Where the system is computationally intensive and it is possible to separate the processing required into a large number of independent computations.
    - Where the system primarily involves the exchange of information between individual computers on a network and there is no need for this information to be centrally stored or managed.
  + P2P networks can be decentralized or semi centralized
    - Decentralized
      * Pros
        + It is highly redundant and therefore both fault-tolerant and tolerant of nodes disconnecting from the network
    - Semi Centralized
      * The role of the server is to help establish contact between peers in the network, or to coordinate the results of a computation.
        + Pros

Enables more control

* + Pros of P2P
    - Allows for effective use of capacity across a network
  + Cons of P2P
    - Security and Trust
      * P2P involves opening you computer to direct interaction with other peers and this means that these systems could, potentially, access any of your resources.
      * Nodes in the network may behave in a malicious way

Software as a Service (Section 18.4)

* In Client-Server you may have to install program on the client computer, which communicates with the server, implements client-side functionality and manages the user interface.
* SAAS is a way of deploying application as thin client-server systems, where the client is a web browser.
* The Notion of SAAS involves hosting the software remotely and providing access to it over the internet. The key elements of SAAS are:
  + Software is deployed on a server (or more commonly a number of servers) and is accessed through a web browser. It is not deployed on a local PC.
  + The software is owned and managed by a software provider, rather than the organizations using the software.
  + Users may pay for the software according to the amount of use they make out of it or through an annual or monthly subscription. Sometimes, the software is free but users then agree to accept advertisements, which fund the software service.
* Benefits for Users
  + Costs of management of software are transferred to the provider
    - The provider is responsible for fixing bugs and installing software upgrades, dealing with changes to the operating system, and ensuring that hardware capacity can meet demand.
  + If someone has multiple computers, there is no need to license software for all of them.
  + If the service is sometimes used, then the pay-per-use model may be cheaper than buying an application
  + The application can be accessed from mobile devices
* Disadvantages
  + Cost of data transfer to the remote service
  + Transfer of data can be slow
  + You may have to pay the service provider according to the amount transferred
  + Lack of control over software evolution (the provider may change the software when they wish)
  + Problems with laws and regulations
  + Differences between SAAS and Service-Oriented Architecture (discussed in chapter 19)
    - SAAS is a way of providing functionality on a remote server with client access through a web browser. The serer maintains the user’s data and state during an interaction session. Transactions are usually long transaction (e.g., editing a document)
    - SOA is an approach to structuring a software system as a set of separate, stateless services, These may be provided by multiple providers and may be distributed. Typically, transactions are short transactions where a service is called, does something, and then returns a result.
    - SAAS is a way of delivering application functionality to users, whereas SOA is an implementation strategy for application systems
    - The functionality implemented using SOA need not appear to users as services.
    - User services do not have to be implemented using SOA
* If SAAS is implemented using SOA, it becomes possible for applications to use service APIs to access the functionality of other applications. They can then be integrated into more complex systems. These are called Mashups and represent another approach to software reuse and rapid software development.
* Service Construction is not usually driven by user requirements, but by the service provider’s assumptions about what users need. -> The software needs to be able to evolve quickly after the provider gets feedback from users on their requirements. Agile Development with incremental delivery is therefore a commonly used approach for software that is to be deployed as a service.
* When you are implementing SAAS, you have to take into account that you may have users of the software from several different organizations. You must take3 factors into account
  + Configurability
    - How do you configure the software for the specific requirements of each organization?
    - The notion of Product Line Architectures (chapter 16) is one way of configuring software for users who have overlapping but not identical requirements. You start with a generic system and adapt this according to the specific requirements of each user.
      * But this does not work for SAAS because it would mean deploying a different copy of the services for each organization that uses the software. Rather, you need to design configurability into the system and provide a configuration interface that allows users to specify their preferences. You then use these to adjust behavior of the software dynamically as it is used. Configuration facilities may include
  + Multi-tenancy
    - How do you present each user of the software with the impression that they are working with their own copy of the system while, at the same time, making efficient use of system resources
      * Problems
        + Data Management

Simplest form of data management is for each customer to have their own database, which they may use and configure as they wish. But, this requires the service provider to maintain many different database instances (one per customer) and to make these available on demand. This is inefficient in terms of server capacity and increases the overall cost of the service

To solve this, you can use a single database with different users being virtually isolated within that database.

* + Scalability
    - How do you design the system so that it can be scaled to accommodate an unpredictably large number of users?
      * Branding
        + Where users from each organization, are presented with an interface that reflects their own organization
      * Business Rules and Workflows
        + Where each organization defines its own rules that govern the use of the service and its data
      * Database Extensions
        + Where each organization defines how the generic services data model is extended to meet its specific needs
      * Access Control
        + Where service customers create individual accounts for their staff and define the resources and functions that are accessible to each of their users.
    - When you consider scalability of SAAS, you are considering “scaling out” rather than “scaling up”.
      * Guidelines for implementing scalable software
        + Develop application where each component is implemented as a simple stateless service that may be run on any server. In the course of a single transaction, a user may therefore interact with instances of the same service that are running on several different servers
        + Design the system using asynchronous interaction so that the application does not have to wait for the result of an interaction (Such as a read request). This allows the application to carry on doing useful work while it is waiting for the interaction to finish
        + Manage resources, such as network and database connections, as a pool so that no single server is likely to run out of resources
        + Design your database to allow fine-grain locking. That is, do not lock out whole records in the database when only part of a record is in use.