CS 530 - Advanced Software Engineering

Distributed Software Engineering

Reference: Sommerville, Software Engineering, 10 ed., Chapter 17

The big picture

Virtually all large computer-based systems are now distributed systems. A distributed system is "a collection of independent computers that appears to the user as a single coherent system." Information processing is distributed over several computers rather than confined to a single machine. Distributed software engineering is therefore very important for enterprise computing systems.

Benefits of distributed systems:

- **Resource sharing:** sharing of hardware and software resources.
- **Openness:** use of equipment and software from different vendors.
- **Concurrency:** concurrent processing to enhance performance.
- **Scalability:** increased throughput by adding new resources.
- **Fault tolerance:** the ability to continue in operation after a fault has occurred.

Distributed systems

Distributed systems are more complex than systems that run on a single processor. Complexity arises because different parts of the system are independently managed as is the network. There is no single authority in charge of the system so top-down control is impossible.

Design issues in distributed systems engineering:

Transparency: to what extent should the distributed system appear to the user as a single system?

Ideally, users should not be aware that a system is distributed and services should be independent of distribution characteristics. In practice, this is impossible because parts of the system are independently managed and because of network delays. Often better to make users aware of distribution so that they can cope with problems. To achieve transparency, resources should be abstracted and addressed logically rather than physically. Middleware maps logical to physical resources.

Openness: should a system be designed using standard protocols that support interoperability?

Open distributed systems are systems that are built according to generally accepted standards. Components from any supplier can be integrated into the system and can inter-operate with the other system components. Openness implies that system components can be independently developed in any programming language and, if these conform to standards, they will work with other components. Web service standards for service-oriented architectures were developed to be open standards.

Scalability: how can the system be constructed so that it is scaleable?

The scalability of a system reflects its ability to deliver a high quality service as demands on the system increase:

- **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 its 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.

There is a distinction between scaling-up and scaling-out. Scaling up is more powerful system; scaling out is more system instances.

Security: how can usable security policies be defined and implemented?

When a system is distributed, the number of ways that the system may be attacked is significantly increased, compared to centralized systems. If a part of the system is successfully attacked then the attacker may be able to use this as a 'back door' into other parts of the system. Difficulties in a distributed system arise because different organizations may own parts of the system. These organizations may have mutually incompatible security policies and security mechanisms. The types of attack that a distributed system must defend itself against are:

- **Interception,** where communications between parts of the system are intercepted by an attacker so that there is a loss of confidentiality.
- **Interruption,** where system services are attacked and cannot be delivered as expected.
- **Denial of service** attacks involve bombarding a node with illegitimate service requests so that it cannot deal with valid requests.
- **Modification,** where data or services in the system are changed by an attacker.
- **Fabrication,** where an attacker generates information that should not exist and then uses this to gain some privileges.

Quality of service: how should the quality of service be specified?

The quality of service (QoS) offered by a distributed system reflects the system's ability to deliver its services dependably and with a response time and throughput that is acceptable to its users. Quality of service is particularly critical when the system is dealing with time-critical data such as sound or video streams. In these circumstances, if the quality of service falls below a threshold value then the sound or video may become so degraded that it is impossible to understand.

Failure management: how can system failures be detected, contained and repaired?

In a distributed system, it is inevitable that failures will occur, so the system has to be designed to be resilient to these failures. "You know that you have a distributed system when the crash of a system that you've never heard of stops you getting any work done." Distributed systems should include mechanisms for discovering if a component of the system has failed, should continue to deliver as many services as possible in spite of that failure and, as far as possible, automatically recover from the failure.

Two types of interaction between components in a distributed system:

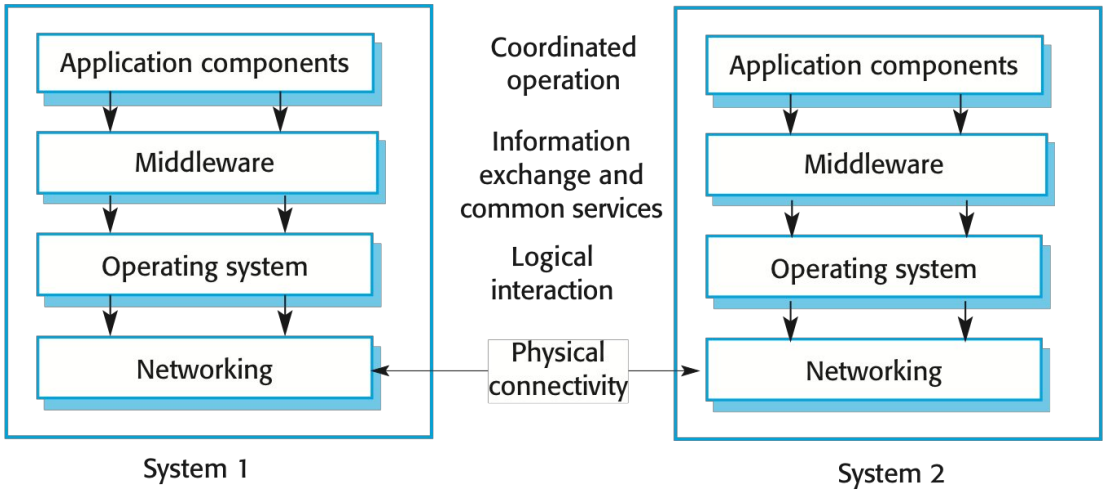
Procedural interaction, where one computer calls on a known service offered by another computer and waits for a response.

Procedural communication in a distributed system is implemented using remote procedure calls (RPC). In a remote procedure call, one component calls another component as if it was a local procedure or method. The middleware in the system intercepts this call and passes it to a remote component. This carries out the required computation and, via the middleware, returns the result to the calling component. A problem with RPCs is that the caller and the callee need to be available at the time of the communication, and they must know how to refer to each other.

Message-based interaction, involves the sending computer sending information about what is required to another computer. There is no necessity to wait for a response.

Message-based interaction normally involves one component creating a message that details the services required from another component. Through the system middleware, this is sent to the receiving component. The receiver parses the message, carries out the computations and creates a message for the sending component with the required results. In a message-based approach, it is not necessary for the sender and receiver of the message to be aware of each other. They simple communicate with the middleware.

Middleware is software that can manage diverse components of a distributed system, and ensure that they can communicate and exchange data. The components in a distributed system may be implemented in different programming languages and may execute on completely different types of processor. Models of data, information representation and protocols for communication may all be different.



Interaction support, where the 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. The provision of common services, where the 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 inter-operate and provide user services in a consistent way.

Client-server computing

Distributed systems that are accessed over the Internet are normally organized as client-server systems. In a client-server system, the user interacts with a program running on their local computer (e.g. a web browser or mobile application). This interacts with another program running on a remote computer (e.g. a web server). The remote computer provides services, such as access to web pages, which are available to external clients.

Layers in a client/server system:

- **Presentation** is concerned with presenting information to the user and managing all user interaction.
- **Data handling** manages the data that is passed to and from the client. Implement checks on the data, generate web pages, etc.
- **Application processing layer** is concerned with implementing the logic of the application and so providing the required functionality to end users.
- **Database** stores data and provides transaction management services, etc.

Architectural patterns for distributed systems

Common **architectural patterns** for organizing the architecture of a distributed system:

Master-slave architecture

Master-slave architectures are commonly used in real-time systems in which guaranteed interaction response times are required. There may be separate processors associated with data acquisition from the system's environment, data processing and computation and actuator management. The '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.

Two-tier client-server architecture

In a two-tier client-server architecture, the system is implemented as a single logical server plus an indefinite number of clients that use that server. **Thin-client** model, where the presentation layer is implemented on the client and all other layers (data management, application processing and database) are implemented on a server. **Fat-client** model, where some or all of the application processing is carried out on the client. Data management and database functions are implemented on the server. Distinction between thin and fat client architectures has become blurred. Javascript allows local processing in a browser so 'fat-client' functionality available without software installation. Mobile apps carry out some local processing to minimize demands on network. Auto-update of apps reduces management problems. There are now very few thin-client applications with all processing carried out on remote server.

Multi-tier client-server architecture

In a multi-tier client-server architecture, the different layers of the system, namely presentation, data management, application processing, and database, are separate processes that may execute on different processors. This avoids problems with scalability and performance if a thin-client two-tier model is chosen, or problems of system management if a fat-client model is used. Used when there is a high volume of transactions to be processed by the server.

Distributed component architecture

There is no distinction in a distributed component architecture between clients and servers. Each distributable entity is a component that provides services to other components and receives services from other components. Component communication is through a middleware system. Used when resources from different systems and databases need to be combined, or as an implementation model for multi-tier client-server systems. Benefits include:

- It allows the system designer to delay decisions on where and how services should be provided.
- It is a very open system architecture that allows new resources to be added as required.
- The system is flexible and scalable.
- It is possible to reconfigure the system dynamically with objects migrating across the network as required.

Distributed component architectures suffer from two major disadvantages:

- They are more complex to design than client-server systems. Distributed component architectures are difficult for people to visualize and understand.
- Standardized middleware for distributed component systems has never been accepted by the community. Different vendors, such as Microsoft and Sun, have developed different, incompatible middleware.

As a result of these problems, service-oriented architectures are replacing distributed component architectures in many situations.

Peer-to-peer architecture

Peer to peer (p2p) systems are decentralised systems where computations may be carried out by any node in the network. The overall system is designed to take advantage of the computational power and storage of a large number of networked computers. Most p2p systems have been personal systems but there is increasing business use of this technology. Used when clients exchange locally stored information and the role of the server is to introduce clients to each other. Examples:

- File sharing systems based on the BitTorrent protocol
- Messaging systems such as Jabber
- Payments systems, e.g. Bitcoin
- Databases, e.g. Freenet is a decentralized database
- Phone systems, e.g. Viber
- Computation systems, e.g. SETI@home

P2P architectures are used when

- A system is computationally-intensive and it is possible to separate the processing required into a large number of independent computations.

- A 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.

Security issues:

- Security concerns are the principal reason why p2p architectures are not widely used.
- The lack of central management means that malicious nodes can be set up to deliver spam and malware to other nodes in the network.
- P2P communications require careful setup to protect local information and if not done correctly, then this is exposed to other peers.

Software as a service

Software as a service (SaaS) involves hosting the software remotely and providing access to it over the Internet. 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 of it or through an annual or monthly subscription. Example: Google Docs. Key elements of SaaS include:

- 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 of it or through an annual or monthly subscription. Sometimes, the software is free for anyone to use but users must then agree to accept advertisements, which fund the software service.

Software as a service (SaaS) and service-oriented architectures (SOA) are related, but they are not the same. Software as a service is a way of providing functionality on a remote server with client access through a web browser. The server maintains the user's data and state during an interaction session. Transactions are usually long transactions e.g. editing a document. Service-oriented architecture 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 then returns a result.

Implementation factors for SaaS:

Configurability

How do you configure the software for the specific requirements of each organization? Service configuration includes:

- 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 service 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.

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?

- Multi-tenancy is a situation in which many different users access the same system and the system architecture is defined to allow the efficient sharing of system resources.
- It must appear to each user that they have the sole use of the system.
- Multi-tenancy involves designing the system so that there is an absolute separation

between the system functionality and the system data.

Scalability

How do you design the system so that it can be scaled to accommodate an unpredictably large number of users?

- Develop applications where each component is implemented as a simple stateless service that may be run on any server.
- 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).
- 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.