Chapter two

2. System Models (or Architectures)

Distributed systems are often complex pieces of software of which the components are by definition dispersed across multiple machines. To master their complexity, it is crucial that these systems are properly organized. There are different ways on how to view the organization of a distributed system, but an obvious one is to make a distinction between the logical organization of the collection of software components and on the other hand the actual physical realization.

The organization of distributed systems is mostly about the software components that constitute the system. These software architectures tell us how the various software components are to be organized and how they should interact. In this chapter we will first pay attention to some commonly applied approaches toward organizing (distributed) computer systems.

The actual realization of a distributed system requires that we instantiate and place software components on real machines. There are many different choices that can be made in doing so. The final instantiation of a software architecture is also referred to as a system architecture.

2.1. ARCHITECTURAL STYLES

Research on software architectures has matured considerably and it is now commonly accepted that designing or adopting an architecture is crucial for the successful development of large systems. Several styles have by now been identified, of which the most important ones for distributed systems are:

1. Layered architectures
2. Object-based architectures
3. Data-centered architectures
4. Event-based architectures

The basic idea for the layered style is simple: components are organized in a layered fashion where a component at layer L; is allowed to call components at the underlying layer L_i, but not the other way around, as shown in Fig. 2-I(a). This model has been widely adopted by the networking community; A key observation is that control generally flows from layer to layer: requests go down the hierarchy whereas the results flow upward.

Organization also followed in object-based architectures, which are illustrated in Fig. 2-1 (b). In essence, each object corresponds to what we have defined as a component, and these components are connected through a (remote) procedure call mechanism. Not surprisingly, this software architecture matches the client-server system architecture we described above. The layered and object-based architectures still form the most important styles for large software systems.

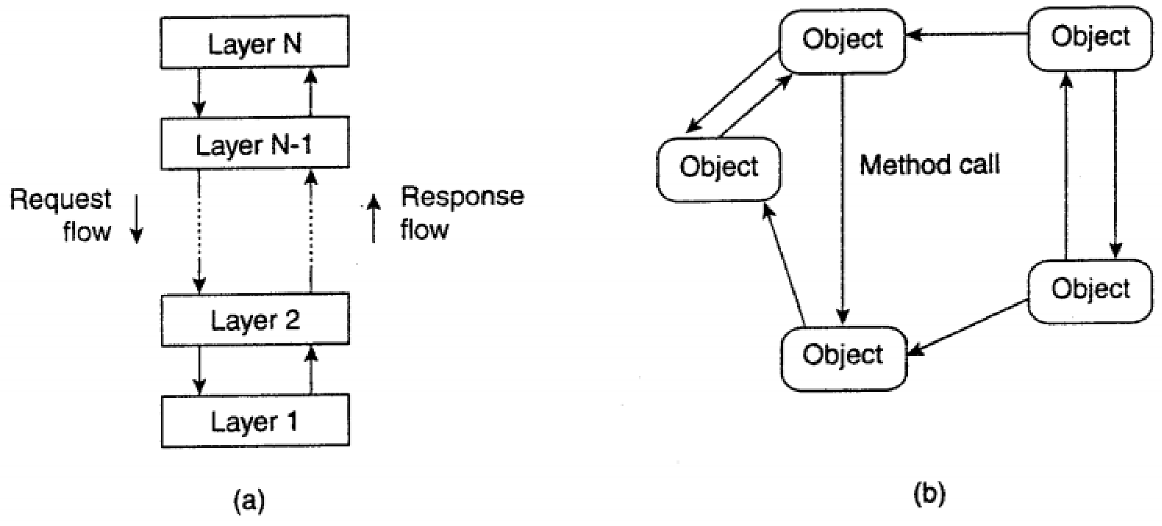

Figure 2-1. The (a) layered and (b) object-based architectural style.

Data-centered architectures evolve around the idea that processes communicate through a common (passive or active) repository. It can be argued that for distributed systems these architectures are as important as the layered and object-based architectures. For example, a wealth of networked applications has been developed that rely on a shared distributed file system in which virtually all communication takes place through files. Likewise, Web-based distributed systems are largely data-centric: processes communicate through the use of shared Web-based data services.

In event-based architectures, processes essentially communicate through the propagation of events, which optionally also carry data, as shown in Fig. 2-2(a). For distributed systems, event propagation has generally been associated with what are known as publish/subscribe systems. The basic idea is that processes publish events after which the middleware ensures that only those processes that subscribed to those events will receive them. The main advantage of event-based systems is that processes are loosely coupled. In principle, they need not explicitly refer to each other. This is also referred to as being decoupled in space, or referentially decoupled.

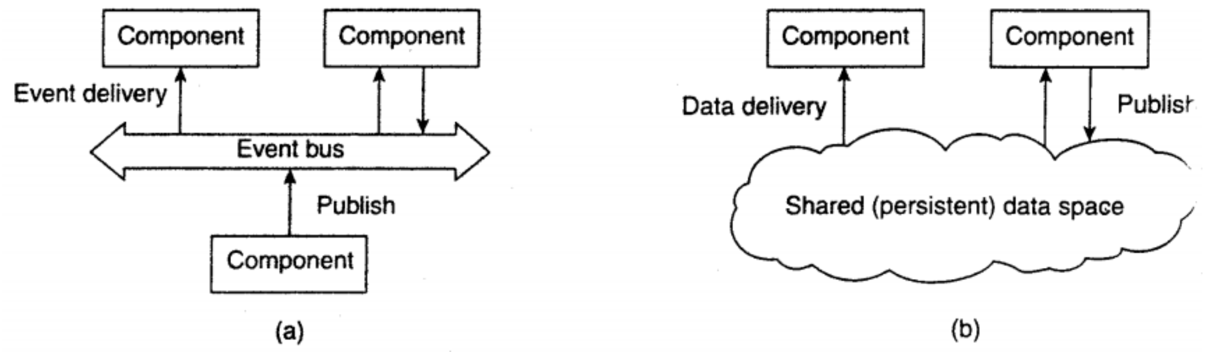

Figure 2-2. The (a) event-based and (b) shared data-space architectural style.

2.2. SYSTEM ARCHITECTURES

Now that we have briefly discussed some common architectural styles, let us take a look at how many distributed systems are actually organized by considering where software components are placed. Deciding on software components, their interaction, and their placement leads to an instance of a software architecture, also called **a system architecture.**

2.2.1. Centralized Architectures

Despite the lack of consensus on many distributed systems issues, there is one issue that many researchers and practitioners agree upon: thinking in terms of clients that request services from servers helps us understand and manage the complexity of distributed systems and that is a good thing.

In the basic client-server model, processes in a distributed system are divided into two (possibly overlapping) groups. A **server** is a process implementing a specific service, for example, a file system service or a database service. A **client** is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply. This client-server interaction, also known as **request-reply behavior** is shown in Fig. 2-3.

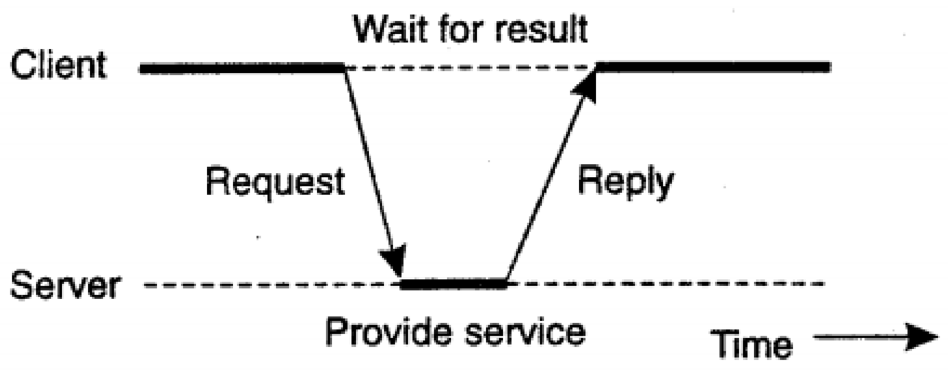

Figure 2-3. General interaction between a client and a server.

Communication between a client and a server can be implemented by means of a simple connectionless protocol when the underlying network is fairly reliable as in many local-area networks. In these cases, when a client requests a service, it simply packages a message for the server, identifying the service it wants, along with the necessary input data. The message is then sent to the server. The latter, in turn, will always wait for an incoming request, subsequently process it, and pack-age the results in a reply message that is then sent to the client.

Using a connectionless protocol has the obvious advantage of being efficient. As long as messages do not get lost or corrupted, the request/reply protocol just sketched works fine. Unfortunately, making the protocol resistant to occasional transmission failures is not trivial. The only thing we can do is possibly let the client resend the request when no reply message comes in.

As an alternative, many client-server systems use a reliable connection-oriented protocol. Although this solution is not entirely appropriate in a local-area network due to relatively low performance, it works perfectly tine in wide-area systems in which communication is inherently unreliable. For example, virtually all Internet application protocols are based on reliable TCP/IP connections.

i. Application Layering

The client-server model has been subject to many debates and controversies over the years. One of the main issues was how to draw a clear distinction be-tween a client and a server. Not surprisingly, there is often no clear distinction. For example, a server for a distributed database may continuously act as a client because it is forwarding requests to different file servers responsible for implementing the database tables. In such a case, the database server itself essentially does no more than process queries. However, considering that many client-server applications are targeted toward supporting user access to databases, many people have advocated a distinction be-tween the following three levels:

1. ***The user-interface level:*** contains all that is necessary to directly interface with the user, such as display management.
2. ***The processing level:*** typically contains the applications
3. ***The data level:*** manages the actual data that is being acted on

ii. Multitiered Architectures

The distinction into three logical levels as discussed so far, suggests a number of possibilities for physically distributing a client-server application across several machines. The simplest organization is to have only two types of machines:

1. A client machine containing only the programs implementing (part of) the user-interface level
2. A server machine containing the rest, that is the programs implementing the processing and data level

In this organization everything is handled by the server while the client is essentially no more than a dumb terminal, possibly with a pretty graphical interface. One approach for organizing the clients and servers is to distribute the pro-grams in the application layers of the previous section across different machines, as shown in Fig. 2-4

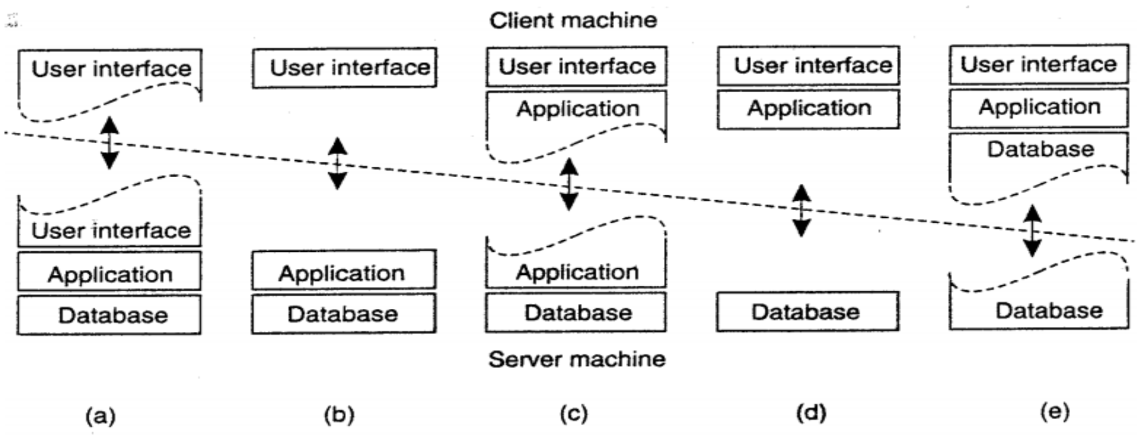

Figure 2-4 Alternative client-server organizations (a)-(e).

One possible organization is to have only the terminal-dependent part of the user interface on the client machine, as shown in Fig. 2-4(a), and give the applications remote control over the presentation of their data.

An alternative is to place the entire user-interface software on the client side, as shown in Fig. 2-4(b). In such cases, we essentially divide the application into a graphical front end, which communicates with the rest of the application (residing at the server) through an application-specific protocol. In this model, the front end (the client software) does no processing other than necessary for presenting the application's interface.

2.2.2. Decentralized Architectures (peer-to-peer)

Multitiered client-server architectures are a direct consequence of dividing applications into a user-interface, processing components, and a data level. The different tiers correspond directly with the logical organization of applications. In many business environments, distributed processing is equivalent to organizing a client-server application as a multitiered architecture.

In modem architectures, it is often the distribution of the clients and the servers that counts, which we refer to as horizontal distribution. In this type of distribution, a client or server may be physically split up into logically equivalent parts, but each part is operating on its own share of the complete data set, thus balancing the load. In this section we will take a look at a class of modern system architectures that support horizontal distribution, known as ***peer-to-peer systems***. Peer-to-peer systems can be grouped into **structured** and **unstructured.**

1. **Structured peer-to-peer architecture**

In a structured peer-to-peer architecture, the overlay network is constructed using a deterministic procedure. By far the most-used procedure is to organize the processes through a distributed hash table (DHT). In a DHT -based system, data items are assigned a random key from a large identifier space, such as a 128-bit or 160-bit identifier. Likewise, nodes in the system are also assigned a random number from the same identifier space.

The crux of every DHT-based system is then to implement an efficient and deterministic scheme that uniquely maps the key of a data item to the identifier of a node based on some distance metric. Most importantly, when looking up a data item, the network address of the node responsible for that data item is returned. Effectively, this is accomplished by routing a request for a data item to the responsible node.

2. Unstructured Peer-to- Peer Architectures

Unstructured peer-to-peer systems largely rely on randomized algorithms for constructing an overlay network. The main idea is that each node maintains a list of neighbors, but that list is constructed in a more or less random way. Like-wise, data items are assumed to be randomly placed on nodes. As a consequence, when a node needs to locate a specific data item, the only thing it can effectively do is flood the network with a search query

One of the goals of many unstructured peer-to-peer systems is to construct an overlay network that resembles a random graph. The basic model is that each node maintains a list of c neighbors, where, ideally, each of these neighbors represents a randomly chosen live node from the current set of nodes. The list of neighbors is also referred to as a ***partial view***.

End of Chapter two