Modularity. Modularity is a basic concept in the design of man-made systems. A complex system is

made of components, or modules, with well-defined functions. Modularity supports the separation of

concerns, encourages specialization, improves maintainability, reduces costs, and decreases the development

time of a system. Hence, it is no surprise that the hardware as well as the software systems are

composed of modules that interact with one another through well-defined interfaces.

In this section we are only concerned with software modularity.We distinguish soft modularity from

enforced modularity. The former means to divide a program into modules that call each other and

communicate using shared memory or follow the procedure call convention. The steps involved in the

transfer of the flowof control between the caller and the callee are: (i) The caller saves its state, including

the registers, the arguments, and the return address, on the stack; (ii) the callee loads the arguments

from the stack, carries out the calculations, and then transfers control back to the caller; (iii) the caller

adjusts the stack, restores its registers, and continues its processing.

Soft modularity hides the details of the implementation of a module and has many advantages. Once

the interfaces of the modules are defined, the modules can be developed independently, and a module

can be replaced with a more elaborate or a more efficient one as long as its interfaces with the other

modules are not changed. The modules can be written using different programming languages and can

be tested independently.

Softmodularity presents a number of challenges. It increases the difficulty of debugging; for example,

a call to a module with an infinite loop will never return. There could be naming conflicts and wrong

context specifications. The caller and the callee are in the same address space and may misuse the stack

(e.g., the callee may use registers that the caller has not saved on the stack, and so on). A strongly typed

language may enforce soft modularity by ensuring type safety at compile or at run time, it may reject

operations or function classes that disregard the data types, or it may not allow class instances to have

their classes altered. Soft modularity may be affected by errors in the run-time system, errors in the

compiler, or by the fact that different modules are written in different programming languages.

TheClient-ServerParadigm. The ubiquitous client-server paradigm is based on enforced modularity;

this means that themodules are forced to interact only by sending and receiving messages. This paradigm

leads to a more robust design where the clients and the servers are independent modules and may fail

separately. Moreover, the servers are stateless; they do not have to maintain state information. The server

may fail and then come up without the clients being affected or even noticing the failure of the server.

The system is more robust since it does not allow errors to propagate. Enforced modularity makes an

attack less likely because it is difficult for an intruder to guess the format of themessages or the sequence

numbers of segments when messages are transported by Transport Control Protocol (TCP).

Last but not least, resources can be managed more efficiently; for example, a server typically consists

of an ensemble of systems: a front-end system that dispatches the requests to multiple back-end systems that process the requests. Such an architecture exploits the elasticity of a computer cloud infrastructure.

The larger the request rate, the larger the number of back-end systems activated.

The client-server paradigm allows systems with different processor architecture (e.g., 32-bit or

64-bit), different operating systems (e.g., multiple versions of operating systems such as Linux, Mac

OS, or MicrosoftWindows), different libraries and other system software to cooperate. The client-server

paradigm increases flexibility and choice; the same service could be available from multiple providers,

or a server may use services provided by other servers, a client may use multiple servers, and so on.

System heterogeneity is a blessing in disguise. The problems it creates outweigh its appeal. It adds

to the complexity of the interactions between a client and a server because it may require conversion

from one data format to another (e.g., from little-endian to big-endian or vice versa), or it may require

conversion to a canonical data representation. There is also uncertainty in terms of response time because

some servers may be more performant than others or may have a lower workload.

A major difference between the basic models of grid and cloud computing is that the former do

not impose any restrictions regarding heterogeneity of the computing platforms. On the other hand, a

computer cloud is a collection of homogeneous systems, systems with similar architecture and running

under the same or very similar system software.

The clients and the servers communicate through a network that itself can be congested. Transferring

large volumes of data through the network can be time consuming; this is a major concern for dataintensive

applications in cloud computing. Communication through the network adds additional delay

to the response time. Security becomes a major concern because the traffic between a client and a server

can be intercepted.

Remote Procedure Call (RPC). RPC is often used for the implementation of client-server systems

interactions. The RPC standard is described in RFC 1831. To use an RPC, a process may use special

services PORTMAP or RPCBIND, available at port 111, to register and for service lookup. RPCmessages

must be well structured; they identify the RPC and are addressed to an RPC demon listening at an RPC

port. XDP is a machine-independent representation standard for RPC.

RPCs reduce the so-called fate sharing between caller and callee but take longer than local calls due

to communication delays. Several RPC semantics are implemented:

• At least once. A message is resent several times and an answer is expected. The server may end

up executing a request more than once, but an answer may never be received. These semantics are

suitable for operations free of side effects.

• At most once. A message is acted on at most once. The sender sets up a time-out for receiving the

response; when the time-out expires, an error code is delivered to the caller. These semantics require

the sender to keep a history of the time stamps of all messages because messages may arrive out of

order. These semantics are suitable for operations that have side effects.

• Exactly once. It implements the at most once semantics and requests an acknowledgment from the

server.

Applications of the Client-Server Paradigm. The large spectrum of applications attests to the role

played by the client-server paradigm in the modern computing landscape. Examples of popular applications

of the client-server paradigm are numerous and include the World Wide Web, the Domain Name

System (DNS), X-windows, electronic mail [see Figure 2.21(a)], event services [see Figure 2.21(b)],

and so on.

TheWorldWideWeb illustrates the power of the client-server paradigm and its effects on society. As

of June 2011 therewere close to 350 millionWeb sites. TheWeb allows users to access resources such as

text, images, digitalmusic, and any imaginable type of information previously stored in a digital format.

A Web page is created using a description language called Hypertext Markup Language (HTML). The

information in each Web page is encoded and formatted according to some standard (e.g., GIF, JPEG

for images, MPEG for videos, MP3 or MP4 for audio, and so on).

The Web is based on a “pull” paradigm; the resources are stored at the server’s site and the client

pulls them from the server. SomeWeb pages are created “on the fly”; others are fetched from disk. The

client, called a Web browser, and the server communicate using an application-level protocol HyperText

Transfer Protocol (HTTP) built on top of the Transport Control Protocol (TCP).

from clients. Figure 2.22 shows the sequence of events when a client sends an HTTP request to a

server to retrieve some information and the server constructs the page on the fly; then it requests an

image stored on the disk. First a TCP connection between the client and the server is established using

a process called a three-way handshake; the client provides an arbitrary initial sequence number in a

special segment with the SYN control bit on. Then the server acknowledges the segment and adds its

own arbitrarily chosen initial sequence number. Finally the client sends its own acknowledgment ACK

as well as the HTTP request, and the connection is established. The time elapsed from the initial request

until the server’s acknowledgment reaches the client is the RTT.

The response time, defined as the time from the instance the first bit of the request is sent until the last

bit of the response is received, consists of several components: the RTT, the server residence time, the

time it takes the server to construct the response, and the data transmission time. RTT depends on the

network latency, the time it takes a packet to cross the network from the sender to the receiver; the data

transmission time is determined by the network bandwidth. In turn, the server residence time depends

on the server load.

Often the client and the server do not communicate directly but through a proxy server, as shown in

Figure 2.23. Proxy servers can provide multiple functions; for example, they may filter client requests and decide whether or not to forward the request based on some filtering rules. The proxy server may

redirect the request to a server in close proximity to the client or to a less loaded server. A proxy can

also act as a cache and provide a local copy of a resource rather than forward the request to the server.

Another type of client-server communication is HTTP tunneling, used most often as a means for

communication from network locations with restricted connectivity. Tunneling means encapsulation of

a network protocol. In our case HTTP acts as a wrapper for the communication channel between the

client and the server (see Figure 2.23).