- **Servers**

A server is a process implementing a specific service on behalf of a collection of clients. In essence, each server is organized in the same way: it waits for an incoming request from a client and subsequently ensures that the request is taken care of, after which it waits for the next incoming request.

There are several ways to organize servers. In the case of an iterative server, the server itself handles the request and, if necessary, returns a response to the requesting client. A concurrent server does not handle the request itself, but passes it to a separate thread or another process, after which it immediately waits for the next incoming request. A multithreaded server is an example of a concurrent server. An alternative implementation of a concurrent server is to fork a new process for each new incoming request. This approach is followed in many UNIX systems. The thread or process that handles the request is responsible for returning a response to the requesting client.

Another issue is where clients contact a server. In all cases, clients send requests to an end point, also called a port, at the machine where the server is running. Each server listens to a specific end point. How do clients know the endpoint of a service? One approach is to globally assign end points for well-known services. For example, servers that handle Internet FTP requests always listen to TCP port 21. Likewise, an HTTP server for the World Wide Web will always listen to TCP port 80. These end points have been assigned by the Internet Assigned Numbers Authority (lANA), and are documented in Reynolds and Postel (1994). With assigned end points, the client only needs to find the network address of the machine where the server is running. As we explain in the next chapter, name services can be used for that purpose.

There are many services that do not require a preassigned end point. For example, a time-of-day server may use an end point that is dynamically assigned to its local operating system. In that case, a client will first have to look up the end point. One solution is to have a special daemon running on each machine that runs servers. The daemon keeps track of the current end point of each service implemented by a co-located server. The daemon itself listens to a well-known end point. A client will first contact the daemon, request the end point, and then contact the specified server, as shown in Fig. 3-11(a).

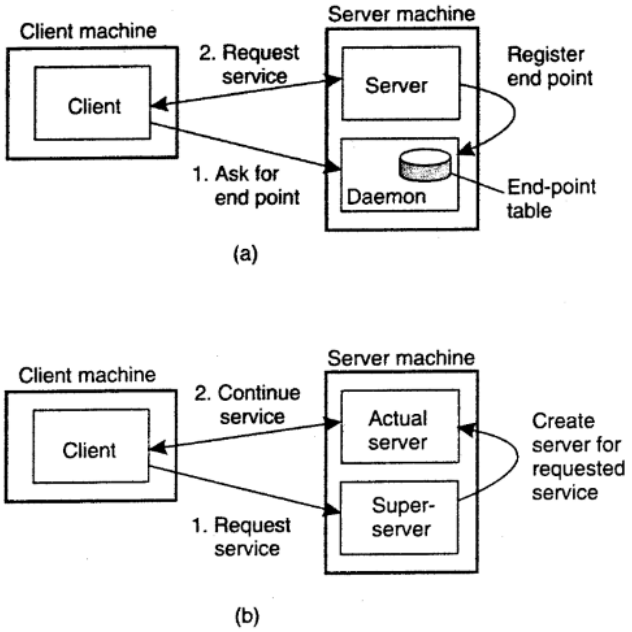


Figure 3-11. (a) Client-to-server binding using a daemon. (b) Client-to-server binding using a superserver..

It is common to associate an end point with a specific service. However, actually implementing each service by means of a separate server may be a waste of resources. For example, in a typical UNIX system, it is common to have lots of servers running simultaneously, with most of them passively waiting until a client request comes in. Instead of having to keep track of so many passive processes, it is often more efficient to have a single superserver listening to each end point associated with a specific service, as shown in Fig. 3-1l(b). This is the approach taken, for example, with the inetd daemon in UNIX. Inetd listens to a number of well-known ports for Internet services. When a request comes in, the daemon forks a process to take further care of the request. That process will exit after it is finished.

Another issue that needs to be taken into account when designing a server is whether and how a server can be interrupted. For example, consider a user who has just decided to upload a huge file to an FTP server. Then, suddenly realizing that it is the wrong file, he wants to interrupt the server to cancel further data transmission. There are several ways to do this. One approach that works only too well in the current Internet (and is sometimes the only alternative) is for the user to abruptly exit the client application (which will

automatically break the connection to the server), immediately restart it, and pretend nothing happened. The server will eventually tear down the old connection, thinking the client has probably crashed.

A much better approach for handling communication interrupts is to develop the client and server such that it is possible to send out-of-band data, which is data that is· to be processed by the server before any other data from that client.

One solution is to let the server listen to a separate control endpoint to which the client sends out-of-band data, while at the same time listening (with a lower priority) to the end point through which the normal data passes. Another solution is to send out-of-band data across the same connection through which the client is sending the original request. In TCP, for example, it is possible to transmit urgent data. When urgent data is received at the server, the latter is interrupted (e.g .• through a signal in UNIX systems), after which it can inspect the data and handle them accordingly.

A final, important design issue, is whether or not the server is stateless. A stateless server does not keep information on the state of its clients, and can change its own state without having to inform any client (Birman, 2005). A Web server, for example, is stateless. It merely responds to incoming HTTP requests, which can be either for uploading a file to the server or (most often) for fetching a file. When the request has been processed, the Web server forgets the client completely. Likewise, the collection of files that a Web server manages (possibly in cooperation with a file server), can be changed without clients having to be informed. .

Note that in many stateless designs, the server actually does maintain information on its clients, but crucial is the fact that if this information is lost, it will not lead to a disruption of the service offered by the server. For example, a Web server generally logs all client requests. This information is useful, for example, to decide whether certain documents should be replicated, and where they should be replicated to. Clearly, there is no penalty other than perhaps in the form of suboptimal performance if the log is lost.

A particular form of a stateless design is where the server maintains what is known as soft state. In this case, the server promises to maintain state on behalf of the client, but only for a limited time. After that time has expired, the server falls back to default behavior, thereby discarding any information it kept on account of the associated client. An example of this type of state is a server promising to keep a client informed about

updates, but only for a limited time.

After that, the client is required to poll the server for updates. Soft-state approaches originate from protocol design in computer networks, but can be equally applied to server design (Clark, 1989; and Lui et al., 2004).

In contrast, a stateful server generally maintains persistent information on its clients. This means that the information needs to be explicitly deleted by the server. A typical example is a file server that allows a client to keep a local copy of a file, even for performing update operations. Such a server would maintain a table containing (client, file) entries. Such a table allows the server to keep track of which client currently has the update permissions on which file, and thus possibly also the most recent version of that file.

This approach can improve the performance of read and write operations as perceived by the client. Performance improvement over stateless servers is often an important benefit of stateful designs. However, the example also illustrates the major drawback of stateful servers. If the server crashes, it has to recover its table of (client, file) entries, or otherwise it cannot guarantee that it has processed the most recent updates on a file. In general, a stateful server needs to recover its entire state as it was just before the crash. As we discuss in Chap. 8, enabling recovery can introduce considerable complexity. In a stateless design, no special measures need to be taken at all for a crashed server to recover. It simply starts running again, and waits for client requests to come in.

What remains for a permanent state is typically information maintained in databases, such as customer information, keys associated with purchased software, etc. However, for most distributed systems, maintaining session state already implies a stateful design requiring special measures when failures do happen and making explicit assumptions about the durability of state stored at the server. When designing a server, the choice for a stateless or stateful design should not affect the services provided by the server. For example, if files have to be opened before they can be read from, or written to, then a stateless server should one way or the other mimic this behavior.

In other cases, a server may want to keep a record on a client's behavior so that it can more effectively respond to its requests. For example, Web servers sometimes offer the possibility to immediately direct a client to his favorite pages.

This approach is possible only if the server has history information on that client. When the server cannot maintain state, a common solution is then to let the client send along

additional information on its previous accesses. In the case of the Web, this information is often transparently stored by the client's browser in what is called a cookie, which is a small piece of data containing client-specific information that is of interest to the server. Cookies are never executed by a browser; they are merely stored.

The first time a client accesses a server, the latter sends a cookie along with the requested Web pages back to the browser, after which the browser safely tucks the cookie away. Each subsequent time the client accesses the server, its cookie for that server is sent along with the request. Although in principle, this approach works fine, the fact that cookies are sent back for safekeeping by the browser is often hidden entirely from users. So much for privacy.

**Server Clusters**

Simply put, a server cluster is nothing else but a collection of machines connected through a network, where each machine runs one or more servers. The server clusters that we consider here, are the ones in which the machines are connected through a local-area network, often offering high bandwidth and low latency.

In most cases, a server cluster is logically organized into three tiers, as shown in Fig. 3-12. The first tier consists of a (logical) switch through which client requests are routed. Such a switch can vary widely. For example, transport-layer switches accept incoming TCP connection requests and pass requests on to one of servers in the cluster, as we discuss below. A completely different example is a Web server that accepts incoming HTTP requests, but that partly passes requests to application servers for further processing only to later collect results and return an HTTP response.

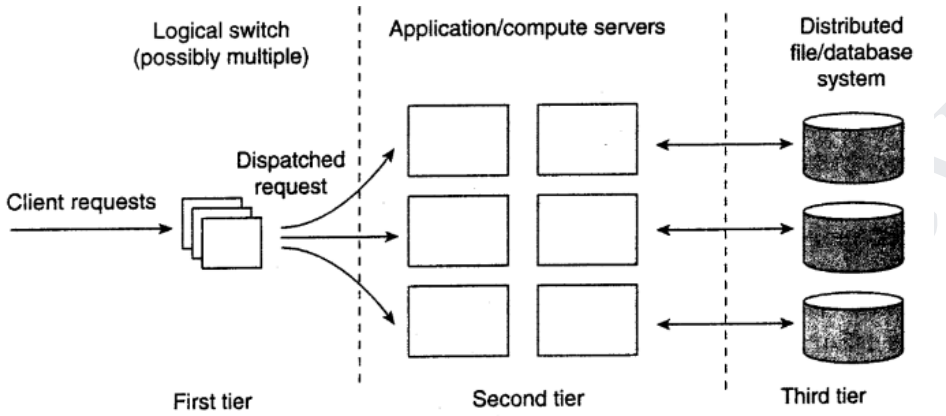


Figure 3-12. The general organization of a three-tiered server cluster.

As in any multitiered client-server architecture, many server clusters also contain servers dedicated to application processing. In cluster computing, these are typically servers running on high-performance hardware dedicated to delivering compute power. However, in the case of enterprise server clusters, it may be the case that applications need only run on relatively low-end machines, as the required compute power is not the bottleneck, but access to storage is.

This brings us the third tier, which consists of data-processing servers, notably file and database servers. Again, depending on the usage of the server cluster, these servers may be running specialized machines, configured for high-speed disk access and having large server-side data caches.

Of course, not all server clusters will follow this strict separation. It is frequently the case that each machine is equipped with its own local storage, often integrating application and data processing in a single server leading to a two-tiered architecture. For example, when dealing with streaming media by means of a server cluster, it is common to deploy a two-tiered system architecture, where each machine acts as a dedicated media server (Steinmetz and Nahrstedt, 2004).

When a server cluster offers multiple services, it may happen that different machines run different application servers. As a consequence, the switch will have to be able to distinguish services or otherwise it cannot forward requests to the proper machines. As it turns out, many second-tier machines run only a single application. This limitation comes from dependencies on available software and hardware, but also that different

applications are often managed by different administrators. The latter do not like to interfere with each other's machines.

As a consequence, we may find that certain machines are temporarily idle, while others are receiving an overload of requests. What would be useful is to temporarily migrate services to idle machines. A solution proposed in Awadallah and Rosenblum (2004), is to use virtual machines allowing a relatively easy migration of code to real machines.

Let us take a closer look at the first tier, consisting of the switch. An important design goal for server clusters is to hide the fact that there are multiple servers. In other words, client applications running on remote machines should have no need to know anything about the internal organization of the cluster. This access transparency is invariably offered by means of a single access point, in turn implemented through some kind of hardware switch such as a dedicated machine.

The switch forms the entry point for the server cluster, offering a single network address. For scalability and availability, a server cluster may have multiple access points, where each access point is then realized by a separate dedicated machine. We consider only the case of a single access point.

A standard way of accessing a server cluster is to set up a TCP connection over which application-level requests are then sent as part of a session. A session ends by tearing down the connection. In the case of transport-layer switches, the switch accepts incoming TCP connection requests, and hands off such connections to one of the servers (Hunt et al, 1997; and Pai et al., 1998). The principle working of what is commonly known as TCP handoff is shown in Fig. 3-13.

When the switch receives a TCP connection request, it subsequently identifies the best server for handling that request, and forwards the request packet to that server. The server, in turn, will send an acknowledgment back to the requesting client. but inserting the switch's IP address as the source field of the header of the IP packet carrying the TCP segment. Note that this spoofing is necessary for the client to continue executing the TCP protocol: it is expecting an answer back from the switch, not from some arbitrary server it has never heard of before. Clearly, a TCP-handoff implementation requires operating-system level modifications.

It can already be seen that the switch can play an important role in distributing the load among the various servers. By deciding where to forward a request to, the switch also

decides which server is to handle further processing of the request. The simplest load-balancing policy that the switch can follow is round robin: each time it picks the next server from its list to forward a request to.

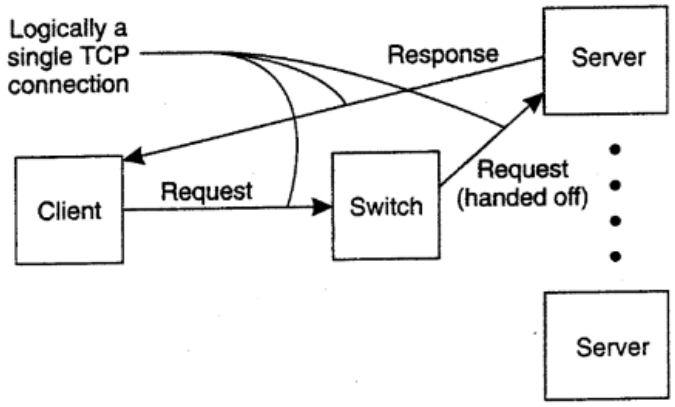


Figure 3-13. The principle ofTCP handoff.

**Distributed Servers**

The server clusters discussed so far are generally rather statically configured. In these clusters, there is often a separate administration machine that keeps track of available servers, and passes this information to other machines as appropriate, such as the switch.

As we mentioned, most server clusters offer a single access point. When that point fails, the cluster becomes unavailable. To eliminate this potential problem, several access points can be provided, of which the addresses are made publicly available. For example, the Domain Name System (DNS) can return several addresses, all belonging to the same host name. This approach still requires clients to make several attempts if one of the addresses fails. Moreover, this does not solve the problem of requiring static access points.

Having stability, like a long-living access point, is a desirable feature from a client's and a server's perspective. On the other hand, it is also desirable to have a high degree of flexibility in configuring a server cluster, including the switch. This observation has led to a design of a distributed server which effectively is nothing but a possibly dynamically changing set of machines, with also possibly varying access points, but which nevertheless- appears to the outside world as a single. powerful machine. The design of

such a distributed server is given in Szy-maniak et al. (2005). We describe it briefly here. The basic idea behind a distributed server is that clients benefit from a robust, high-performing, stable server. These properties can often be provided by high-end mainframes, of which some have an acclaimed mean time between failure of more than 40 years. However, by grouping simpler machines transparently into a cluster, and not relying on the availability of a single machine, it may be possible to achieve a better degree of stability than by each component individually. For example, such a cluster could be dynamically configured from end-user machines, as in the case of a collaborative distributed system.

Let us concentrate on how a stable access point can be achieved in such a system. The main idea is to make use of available networking services, notably mobility support for IP version 6 (MIPv6). In MIPv6, a mobile node is assumed to have a home network where it normally resides and for which it has an associated stable address, known as its home address (HoA). This home network has a special router attached, known as the home agent, which will take care of traffic to the mobile node when it is away. To this end, when a mobile node attaches to a foreign network, it will receive a temporary care-of address (CoA) where it can be reached. This care-of address is reported to the node's home agent who will then see to it that all traffic is forwarded to the mobile node. Note that applications communicating with the mobile node will only see the address associated with the node's home network. They will never see the care-of address.

This principle can be used to offer a stable address of a distributed server. In this case, a single unique contact address is initially assigned to the server cluster. The contact address will be the server's life-time address to be used in all communication with the outside world. At any time, one node in the distributed server will operate as an access point using that contact address, but this role can easily be taken over by another node. What happens is that the access point records its own address as the care-of address at the home agent associated with the distributed server. At that point, all traffic will be directed to the access point, who will then take care in distributing requests among the currently participating nodes. If the access point fails, a simple fail-over mechanism comes into place by which another access point reports a new care-of address.

This simple configuration would make the home agent as well as the access point a potential bottleneck as all traffic would flow through these two machines. This situation can be avoided by using an MIPv6 feature known as route optimization. Route optimization works as follows. Whenever a mobile node with home address HA reports

its current care-of address, say CA, the horne agent can forward CA to a client. The latter will then locally store the pair (HA, CAY· From that moment on, communication will be directly forwarded to CA. Although the application at the client side can still use the horne address, the underlying support software for MIPv6 will translate that address to CA and use that instead.

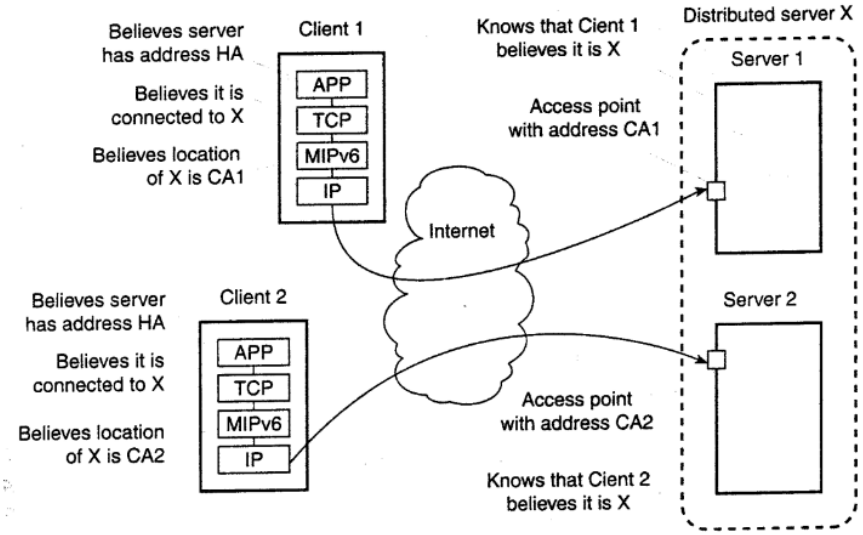


Figure 3-14. Route optimization in a distributed server.

Route optimization can be used to make different clients believe they are communicating with a single server, where, in fact, each client is communicating with a different member node of the distributed server, as shown in Fig. 3-14. To this end, when an access point of a distributed server forwards a request from client C 1 to, say node S 1 (with address CA 1)' it passes enough information to S 1 to let it initiate the route optimization procedure by which eventually the client is made to believe that the care-of address is CA r- This will allow C 1 to store the pair (HA, CA 1)' During this procedure, the access point (as well as the horne. agent) tunnel most of the traffic between C1 and S r- This will prevent the horne agent from believing that the care-of address has changed, so that it will continue to communicate with the access point.

Of course, while this route optimization procedure is taking place, requests from other clients may still corne in. These remain in a pending state at the access point until they can be forwarded. The request from another client C2 may then be forwarded to member node S2 (with address CA 2), allowing the latter to let client Cz store the pair (HA, CA2

). As a result, different clients will be directly communicating with different members of the distributed server, where each client application still has the illusion that this server has address HA. The home agent continues to communicate with the access point talking to the contact address.