Servers

A server is a process implementing a special service on behalf of a collection of clients. In essence, each server is organised 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 an incoming request.

Concurrent vs Iterative servers

Iterative servers

The server itself handles the request and, if necessary, returns a response to the requesting client.

Concurrent Servers

A concurrent server does not handle the request itself, but passes it to a seperate thread or another process, after which it immediately waits for an incoming request. Example: a multithreaded server. The thread or process that handles the request is responsible for returning a response to the requesting client.

Design Issues

Contacting a server: end points (end points = ports)

In all cases, the client send requests to an **end point**, also called a **port**, at the machine where the server is running . Each server listens to a specific port or end point.

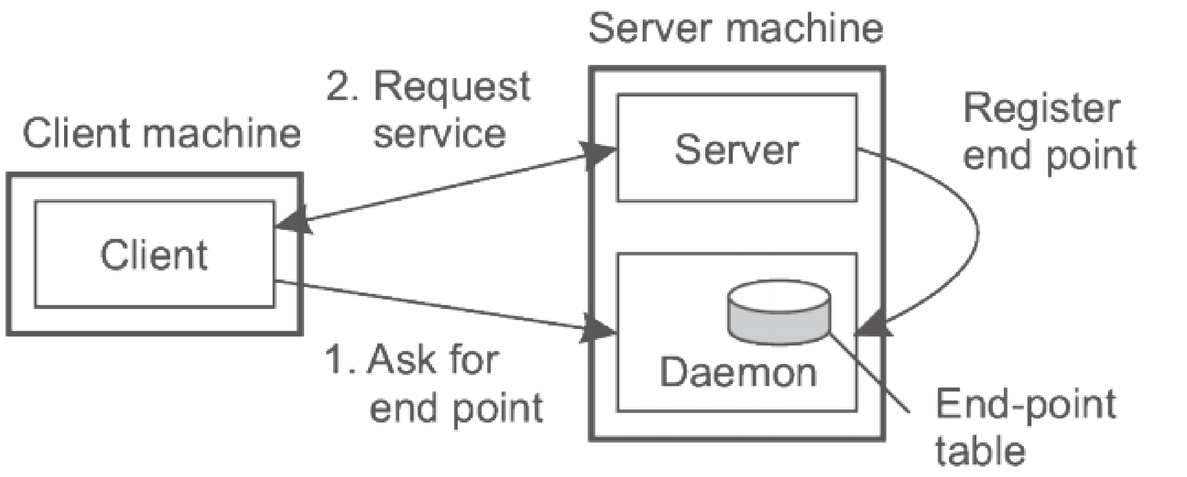
HOW DO CLIENTS KNOW THE ENDPOINT OF A SERVICE?

Well-known services are globally assigned end points. FTP service listens to port TCP 21. Likewise, an HTTP server for world wide web is on TCP port `80`. These ports have assigned by the Internet Assigned Numbers Authority (IANA). With assigned end points, clients only need network address of the server.

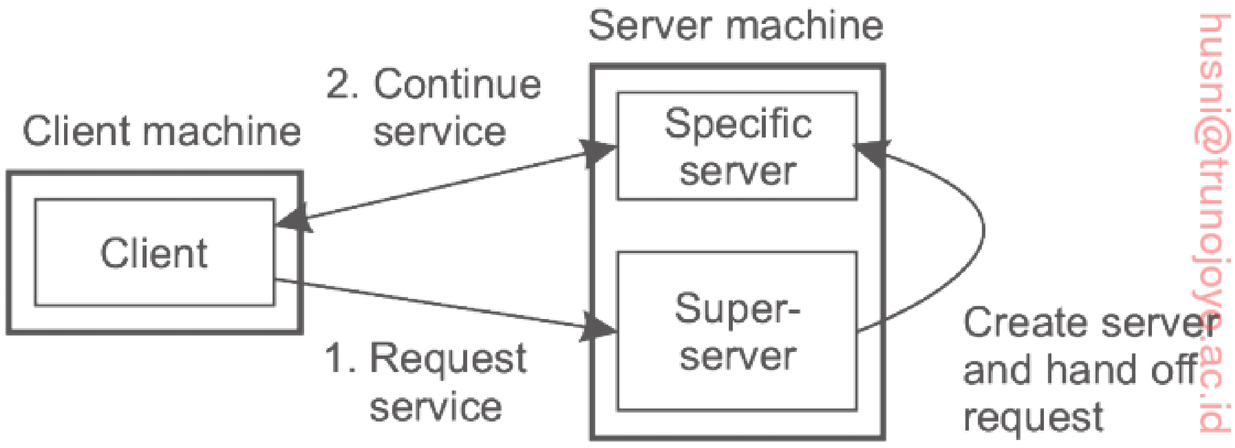
There are many services that do not require preassigned endpoint. For example, a time-of-day server may use a dynamically assigned endpoint assigned to it by its local OS. In that case, the client will have to first look up the endpoint. **SOLUTION**: daemon that runs on each machine that runs server. The daemon keeps track of current endpoint of each service implemented by co-located server.

Superserver

It listens to each endpoint associated with a specific service, an efficient solution to keeping track of many passive processes associating with a different service. Example: the `inetd` daemon in unix, which forks a process to handle an incoming request.



(a)



(b)

Interrupting a Server

Issue: Consider a user who has uploaded a huge file over a FTP server, but now realizing that it is the wrong file, he wants to interrupt the server to cancel the file upload or further data transmission.

Ways to do this:

- The user can abruptly exit the client application, immediately restart it, and pretend that nothing happened. The server will eventually tear down the old connection, thinking the client has crashed.
- A much better approach 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 seperate control end point to which the client sends out-of-band data , while listening to the normal end point. Another solution is to send the out-of-band data across the same connection through which the client is sending the request.

Stateless vs Stateful servers

Stateless servers

A **stateless server** does not keep information on the state of its client, and can change its own state without having to inform any client. A web server, for example, is stateless. The web server forgets the client completely after a request from that same client is completed.

Soft State

In this case, the server promises to maintain state on behalf of its client, but only for a limited time. After that time has passed, the server falls back to its default behavior, thereby discarding any information it kept on the account of the associated client.

Stateful servers

Stateful servers generally maintain persistent information on its clients. This means that the informatiom needs to be explicitly deleted by the server.

Disadvantage of stateful server: If the server crashes, it has to re retrieve the working files again so that they are the most recent versions. Enabling recovery can introduce considerable complexity.

Permanent State vs Session State

Session State: It is associated with a series of operations by a single user and should be maintained for some time, but not indefinitely. Often maintained in three tier client-server architectures, where the application server actually needs to access a database server through a series of queries before being able to respond to the requesting client. No real harm is done, if the client can re-issue the original request.

Permanent State: Information maintained in databases, such as customer information, keys associated with purchased software, etc.

When designing a server, the choice for a stateless or a stateful server should not affect the services offered by the server.

COOKIES

- The additional information of its previous accesses the client sends along.
- It is transparently stored by the client's browser.
- Containing client specific information that is of the interest to the user.
- Never executed by the browser.

The first time a client accesses a server, the server sends a cookie along with requested web pages back to the browser, after which the browser safely tucks the cookies away. Each subsequent time, the cookie for that server is sent along with the request to the server.

Object Servers

The difference between **object servers** and other (traditional) servers is that the object server by itself does not provide a specific service. Specific services are implemented by **objects** that reside in the server. Essentially, the server provides only the means to invoke local objects, based on requests from remote clients. It is relatively easy to change services by simply adding or removing objects.

Acts as a server where objects live.

An object consists of 2 parts:

- Data representing its state,
- code for executing its methods.

There are differences in the ways in which object servers invoke their objects.

For an object to be invoked, the object server needs to know which code to execute, on which data it should operate, whether it should start a seperate thread to take care of the invocation, and so on. A simple approach is to assume that all objects *look alike* and there is only one way to invoke an object. Unfortunately, such an approach is generally inflexible and often unnecessarily constrains developers of distributed objects.

A much better approach is for a server to support different policies.

Transient Object: An object that exists only as long as its server exists, but possibly for a shorter period of time.

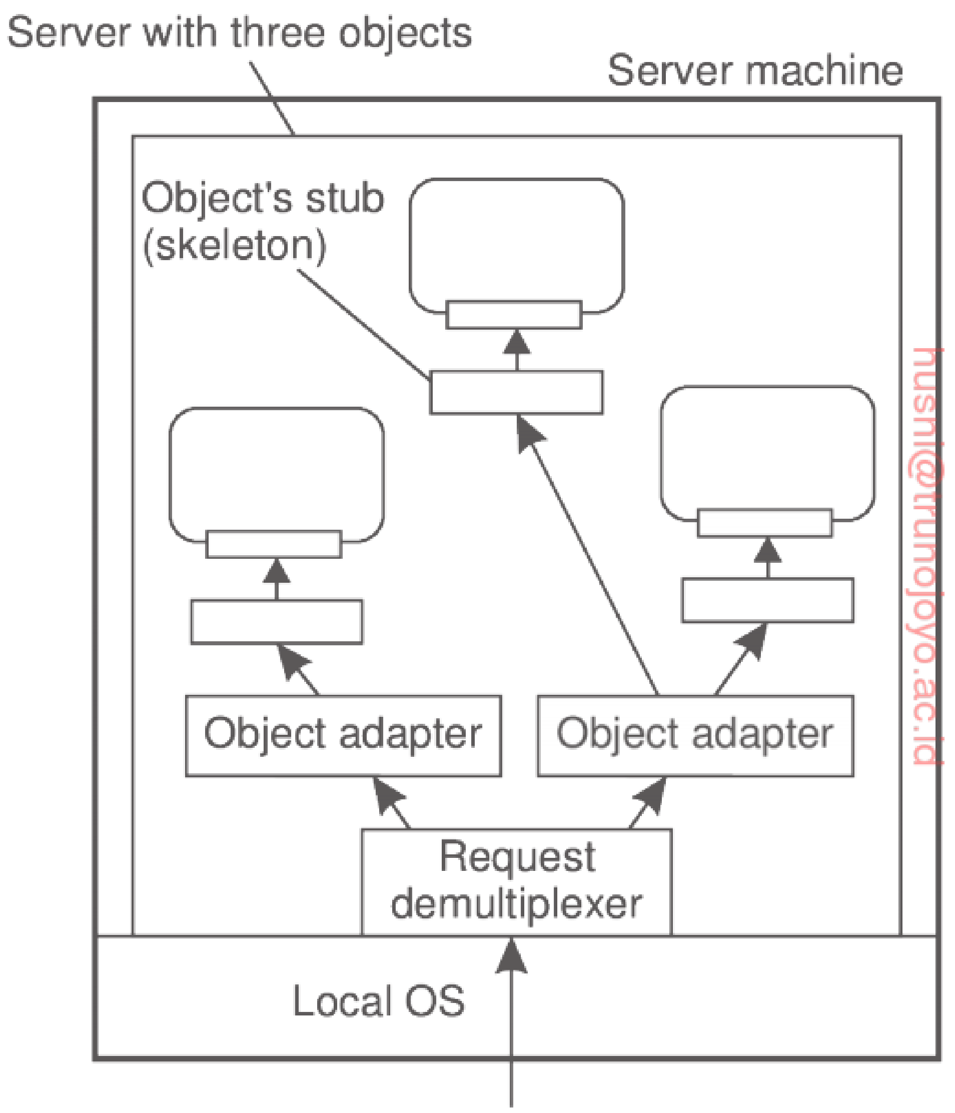
A reasonable policy is to create a transient object at the first invocation request and to destroy it as soon as no clients are bound to it anymore. The **advantage** of this approach is that a transient object will need a server's resources only as long as the object is really needed. The **drawback** is that an invocation may take some time to complete, because the object needs to created first. Therefore, an alternative policy is to create all transient objects at the time server is initialised, at the cost of consuming resources even when no client is making use of the object.

In similar fashion, a server could follow the policy that each of its objects is placed in a memory segment of its own. The alternative approach is to let the objects at least share their code.

Approaches respecting threading: The simplest approach is to implement the server with only a single thread of control. Alternatively, the server may have several threads, one for each of its objects. Whenever an invocation request comes in for an object, the server passes the request to the thread responsible for that object.. If the thread is currently busy, the request is temporarily queued. The **advantage** of this approach is that the objects are automatically protected against concurrent access.

Decisions on how to invoke an object are commonly referred to as **activation policies**, to emphasize that in many cases the object itself must first be brought in the server's address space before it can be invoked.

Object adapter or **Object wrapper**: Grouping objects per policy, can be thought of as a software implementing a specific activation policy. Has one or more objects under its control. DIfferent activation policies are required to be supported by the server, so several object adapters.



An object server supporting different activation policies.

Object adapter are unaware of the specific interfaces of the objects they control. Otherwise, they could never by generic. It can extract an object's reference from an invocation request and then it can dispatch the request to the referenced object obviously.

Servant: General term for a **piece of code** that forms the implementation of an object.