Unit 3. Client-server architectures

Outcomes Summary. You will learn
- how and why servers are usually layered in tiers
- how servers can be made more scalable

Further Reading: CDK2005 2.2.

The client-server model is the most common form of distributing responsibility. The server will typically fully control a resource (data, secret knowledge or algorithm) that needs to be accessed by multiple clients in a controlled way.

A server usually does not do anything until a client sends a request. This is called a *pull* model: clients pull services from a server. In the cases where a server takes the initiative, we talk about the *push* model. One common example of a push activity is where the server *notifies* clients of an event. Clients usually need to subscribe with the server for particular types of events.

Clients are often using several (small number of) servers at the same time. A node can act as a client and as a server at the same time.

The most common example of such dual-role nodes are found in various enterprise systems that embody the so-called 3-tier architecture consisting of:

- *presentation tier* made of user interface clients using
- *application logic tier* made of servers that embody business processes
- *data tier* made of pure servers that store data and perform basic operations upon it.

Sometimes there are more than 3 tiers, each tier embodying a different level of abstraction of the system and allowing a different form of interaction with it.

The simplest client-server architecture is a special case of multi-tiered architecture with only 2 tiers.

3.1 Client-server example

The simple chat system deployed in Practical 1 uses a pure client-server model with a notification service. The server implements one logical chat room and clients can subscribe and unsubscribe for notifications of chat messages. Whether or not a client has subscribed, they can send messages to the room.

Figure 3.1 illustrates how the server and client work using an activity diagram.

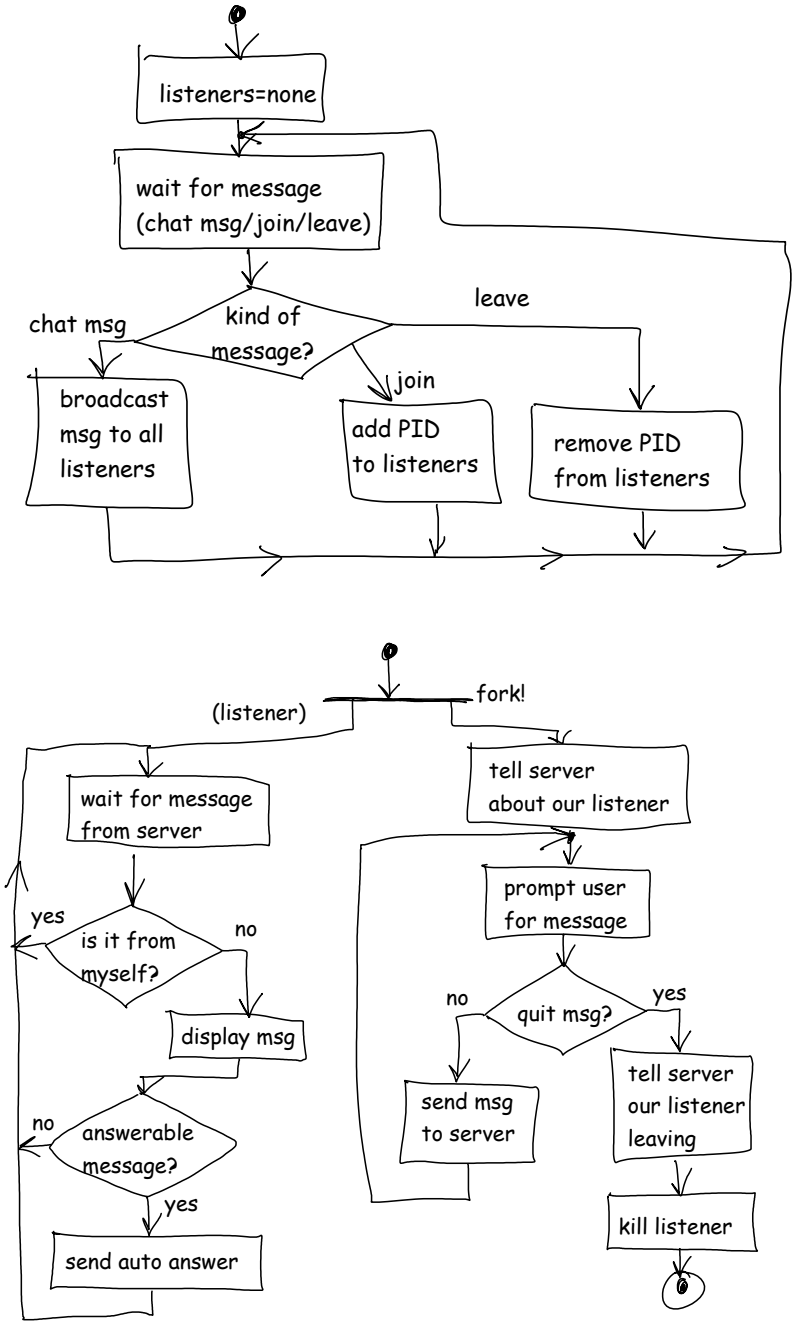

Figure 3.1: Activity of simple chat server and client

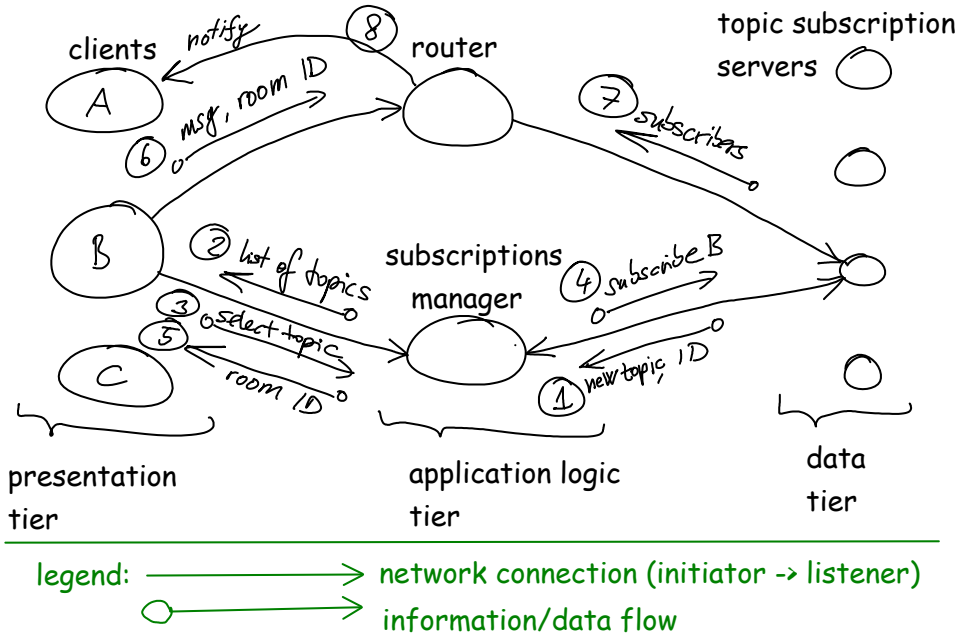

Figure 3.2: A 3-tiered chat system overview

3.2 3-tiered example

The above server supports one chat-room only. Imagine we want many chat-rooms, each for different topics. We could have many instances of the server and the clients would need to know about all the servers and which server does which topic. That would not be very scalable. Ideally the client would only deal with one or two servers which would know about all available topics and be able to subscribe the client to any of them and then allow them to participate in the discussions.

One way to design such a system is to use the 3-tiered approach described earlier. The client is in the presentation tier. The application logic tier is now responsible for managing a list of topics and routing messages between clients. These two responsibilities are independent and it is therefore convenient to split them between two servers: a message router and a room registry. The data tier will hold the information about which client is in which room. It is possible to have one logical node per room, as shown in Figure 3.2.

More precisely, a room subscription server at the data tier is responsible for:

- knowing its topic,
- announcing the topic to the room registry,
- maintaining a list of the IDs of the clients who subscribed to this topic.

Splitting a server into two tiers like this has both advantages and disadvantages:

- advantage: each node is simpler
- advantage: system is easier to maintain and modify (ie more flexible)
- disadvantage: there is more network traffic
- disadvantage: overall slightly more programming to do

3.3 Combating the server bottleneck

The single biggest problem with client-server architecture is how to make the server sufficiently scalable. Let us review the most common tricks to alleviate the problem.

Dividing responsibility. We have seen this in the previous sub-section in two cases:

(A) Splitting the application logic tier into two independent servers.

It is likely that most of the server load relates to message routing and thus by allowing one server to focus on this activity increased the amount of clients the server can handle, albeit only slightly.

On the other hand, the performance of the room management operations under heavy load would improve very substantially thanks to this separation.

(B) Managing each list of members on a separate logical node.

If the network is fast enough, router's membership look-ups become very fast even when the load is very high. The rooms can be automatically distributed on different physical nodes according to load.

Moreover, the membership details can be secured in isolation from the other parts of the system. Thus division of responsibility can improve not only scalability but also security.

Proxies. A *proxy* behaves like a server but it 'cheats' because it simply uses another server to do all its work. A proxy can be useful for security purposes because it can provide a limited version of the server that hides behind it. A proxy can also provide a stable location for a mobile server.

Any server deployed using Internet standards can be proxied.

Caches. A *cache* is like a proxy except that it keeps a record of past communication with clients and when possible, reuses this communication without putting load on the server. A communication can be reused when a client requires exactly the same service that was given in the past and the cache knows that the server would provide the service in exactly the same way again.

One of the simplest examples is a HTTP cache that remembers what the server replied for some of the recent GET requests. When a new GET request comes for the same URL, the cache quickly checks with the server that the resource has not changed and if not, it reuses the saved resource. The load of the HTTP server can reduce dramatically if many of its clients do not access it directly but through one of a large number of HTTP caches. (Each modern Web browser has a HTTP cache built into it to start with.)

In our 3-tiered chat system example, one could cache the replies of the room subscription servers. This would make sense only if:

- that room has a vast amount of traffic (which would probably make it rather useless),

- the membership does not change very often,
- there are multiple routers making use of these servers.

Replication. Sometimes the protocol for using a server is such that each time a client uses it, the messages are slightly different. For example, if using the HTTPs protocol, the messages are encrypted differently for each client. It is impossible to use caches for HTTPs and similar protocols (except at the client end after the messages are decrypted and become ordinary HTTP). Nevertheless, it is possible to *replicate* (aka *mirror*) such servers and all their resources.

The problem is when a resource is changed on one of such replicated servers, it needs to change on all of them, ideally in a synchronised way. Such synchronisation can be expensive. Where the data does not change often or it is acceptable that the replicas are not 100% equivalent at all times, replication is the best way to scale services.

In our 3-tiered chat system example, it would make perfect sense to replicate the routers. It would be harder and less useful to replicate the room registry and the room subscription servers.

Mobile code. Using mobile code, the server can give instructions to the client how to deal with their need themselves, a bit like sending them a DIY recipe. The most common instance of this is when a Web server sends a browser a HTML page that has JavaScript function definitions and event handlers embedded into it. The JavaScript code instructs the browser eg what menu to show when the user clicks on the map canvas etc. The JavaScript functions may contain instructions that initiate further communication with the server behind the scenes, which has been recently made good use of in various AJAX libraries.

In the 3-tiered chat system the router could be significantly relieved if it would be very lazy and send clients a piece of code that would make the client do both the look-up of the list of subscribers from the room subscription server and the sending of their message to the subscribers.

One may wonder whether such a lazy server has any role to play. In a 3-tiered architecture the code in the presentation tier should be independent of the code in the data tier. Thus our chat clients should not know how to talk to the room subscription servers, in fact they should not even know about their existence. This knowledge resides with the servers alone. Consequently, when the data tier is redesigned, the clients do not need to change at all. Mobile code allows this encapsulation while relieving the server from most work.

We already mentioned in Unit 1 that mobile code poses almost insurmountable security challenges and the client and server have to trust each other to some level. In our example, a rogue client could access the list of all subscribers by inspecting the code and observing the data that is used during the execution of the code. A rogue server could include code that scans files on the client's computer searching for passwords or make the client's process loop forever.