Introduction to Interactive Programming
by Lynn Andrea Stein
A Rethinking CS101 Project

Client-server interaction patterns

Chapter Overview

- Who is responsible for getting something from one entity to another?
- What tradeoffs are involved in this decision?

This chapter concerns the ways in which responsibility for (information) transfer can be allocated between the provider and the recipient and the implications of these design decisions. When the service provider takes responsibility for the transfer, it maintains control of its own workload but may overwhelm the recipient. When the recipient initiates the request, the dual situation is in effect. The participants in this relationship are called servers and clients, and client/server architectures are common in modern software design.

What is a client-server interaction?

Sometimes, one (computational) entity has something that another (computational) entity needs. For example, a baker may have cookies, and you may be hungry. In this case, the entity that has the thing--the baker--is called a **server** and the entity that needs the thing--you--is called a **client**. Although these terms are often used without further explanation, you can see from this description that a client and a server are defined with respect to some (computational) need, or **service** (like a cookie).

In the computational world, a server is often something that provides a particular service to other computers connected by a network. For example, it is common for an organization to have a lot of disks on which its members' information is stored, and to have a single machine responsible for providing access to this storage space. This machine is called a **disk server** or a **file server**. Another machine in the same organization might control the public html access for that organization's world-wide web pages. That machine would be the organization's **web server**. Yet another machine might be in charge of electronic mail for the organization: the **mail server**.

In each of these cases, the service is described by what is provided. But it is also important to characterize how the service is provided, in what form, when, by whom, and to whom, whether it is provided once or repeatedly, whether it is provided to one client at a time or to many clients simultaneously, and who is responsible for initiating the transaction. For example, an important message may be transmitted by certified mail, or it may be communicated by announcement over a public address system. These two services may communicate the same information, but they do so in dramatically different ways.

Postal Services: An Example

In this section, we'll look more closely at a particular real-world, non-computational service: the postal system. In doing so, we will see that most of the major properties of client-server interactions are present in familiar transactions.

The main service provided by the postal system is the transmission of physical letters and packages. In this sense, it is perhaps the original mail server. Its clients are the people who send and receive mail through the system. The post office (or, more generally, the postal system) is the server. It may be obvious that the recipient of mail is a client of the postal system: A recipient gets mail delivery from the postal system. Perhaps less intuitively, the sender of the mail is also a client (of another service of) the postal system: the sender relies on the post office to provide the service of transmitting the letter. We will return to this point later.

Figure #. The post office provides parcel transmission and parcel delivery services. Its clients are senders and recipients, respectively. The post office is a server providing the service of transmission of packages.

This example is actually quite rich, illustrating several points about service providers.

1. A server can provide a variety of services

We have already seen two interrelated services that the postal system provides: mail transmission (to mail senders) and mail delivery (to mail recipients).

Actually, each of these is itself an abstraction of several different services. For example, letter transmission is a somewhat different service than parcel transmission. The post office charges different rates depending on the weight and size of the item to be delivered. Similarly, the postal system provides multiple qualities of service for similar items. A letter can be sent air mail or surface mail, overnight or second day or standard delivery. A parcel can be sent first class or book rate. Each of these services is a specialized form of mail transmission, with different costs, time behaviors, and guarantees. All can be described as the same general mail transmission service, but each has slightly different behavior.

Some service specializations don't fit neatly under the same abstraction. For example, a letter can be sent certified; a return receipt can be requested. Requesting a return receipt even changes the contract between the client -- the mail sender -- and the postal system so that their interaction pattern is different. In traditional mail transmission, the client gives the item to the post office and the interaction ends. (Of course, the post office is still obliged to carry out its end of the deal, delivery.) When a return receipt is requested, the transaction does not end here. Instead, delivery involves the post office's obtaining a signature from the recipient. This signature needs to be transmitted back to the sender; only then is the original transmission service complete. This amounts to the postal equivalent of a callback. (See the chapter on Intelligent Objects.)

The same server that provides these transmission and delivery services -- the post office -- also provides a number of other services, some of which may not even seem related. For example, the United States Postal Service sells stamps and money orders. By special arrangement (i.e., the rental of a post office box) it will hold your mail for you. Some post offices will even provide you with a passport. This last service is one provided by the post office acting on behalf of the Passport Agency.

When we talk about a server, then, it is important to distinguish what service that server is providing. In general, it is impossible to talk about a client or server without (at least implicitly) referring to the service provided.

2. You can have more than one provider of a given service

Among the services provided by the post office is the selling of money orders. But if you want to transmit money through the mail, you can also do so using a check. The check is a service provided by a bank, not by the postal system. So, for sending money through the mail, you can turn to a variety of service providers. Each will have its own set of properties: cheap or expensive, secure or less so, available on demand or only during certain hours, etc.

Figure #. Multiple servers may provide similar services. Both the post office and DHL provide parcel transmission services, though these services vary in cost, guarantees of timeliness, and other properties.

Even considering only the delivery service that is the postal system's "core" service, there are still alternative providers. In the United States, package and letter delivery is provided by United Parcel Service, Federal Express, DHL, and many other vendors. Each of these service providers -- servers -- has a different performance profile. For example, some of these parcel delivery servers are faster, provide "better" service, include a variety of guarantees, cost more or less, etc. At different times, you may wish to select a particular server because its properties best match your needs. But even when multiple servers provide the same (or similar) services, any one service instance -- such as sending one particular parcel -- is likely to go through only one provider of a particular service. For example, when you mail your mother's birthday present, you will pick one carrier to deliver it.

3. Services can be layered

We have seen that the post office provides both transmission and delivery services. These two services together can be used as the basis for other service models.

Figure #. Layered Services. The mail order company provides the shirt service to the purchaser by means of the post office delivery system. The solid arrow shows the shirt procurement service, of which the mail order company is the server and the purchaser the client. The dotted arrows show the parcel shipping and parcel delivery services, of which the post office is the server and the mail order company and purchaser are the clients, respectively. The shirt service is implemented in terms of the parcel shipping and delivery services.

Consider a mail order company, such as a clothing vendor. If I want to buy a Hawaiian shirt, I can order it from the mail order company. The company plays the role of the server and I play the role of the client in this shirt-procurement transaction.

But the clothing vendor is not in the business of shipping. How does the clothing vendor get the shirt to me? One answer is that the clothing vendor may use the post office (or some other parcel delivery service) to ship the shirt. In this case, shirt-procurement is **layered** on top of parcel delivery, i.e, relies on parcel delivery to accomplish the transaction.

Real services often work this way. In fact, network services -- the way that one computer communicates with another -- involve many layers of services. When we look more closely at network services in chapter 21, we will examine only the highest levels of these services. We will use network transmission to build still more sophisticated services -- such as a web server or a chat program -- in exactly the same way that the mail order company relies on the postal service to deliver its shirt.

4. Roles are relative to a service

We have seen how a single server can provide many different services (as the post office does) and that a single service may be provided by many different servers (like the various parcel delivery servers). We have also seen how layering makes it possible for one service to be built out of others. Each of these observations provides further illustraton that the role "server" (or "client") is not an absolute one, but is meaningful only relative to a particular service.

We can't, for example, properly say that the post office is a server. We have to specify what service the post office is providing to whom. Of course, we sometimes skip this information when we think that it is obvious from context. But properly, every server is the server of a particular service interaction; every client is a client with respect to a service interaction.

This is particularly important when we're talking about sophisticated interactions in which a single entity can be simultaneously a client and a server. (Not of the same service interaction, of course.) So, for example:

- The mail order company is simultaneously the server of "shirt purchasing" (I'm the client) and the client of the post office's delivery service.
- In most standard retail transactions, the retailer is simultanously the client of the wholesaler (who sells the retailer the goods) and the server to the general public (like me).
- Many interactions are two-way, like barter. For example, one farmer may supply eggs to a second; the second may provide the first with milk. Each farmer is a service provider (server) as well as a service consumer (client).

Note that in any relationship, an entity can either be a client or a server *of a particular service instance,* but not both.

In computational systems, we typically reserve the term server for ongoing service providers, i.e., persistent entities that can be repeatedly called upon to provide services. That is, servers are full-blown computational entities, not simply program segments.

Implementing client-server interactions

As we have seen, a client-server interaction is one in which the server has something at the beginning and the client has it at the end. This "thing" might be quite abstract -- permission to access some data, for example, or the property of being subscribed to a mailing list -- but the idea is that the client wants it and the server can provide it.

In this section we will to focus on the question of *who initiates the service* and the implications of this decision for client-server interactions. There are many different services provided by many different servers, and many different mechanisms to support these services. These include simple procedure call, the use of channels to transmit requests, even aspects of event-driven programming. The issue of who takes responsibility for service initiation exists no matter what service mechanism is used to implement the interaction, and the tradeoffs described here apply to each of these implementations as well.

Client pull

If a client needs something (or some service) from a server, perhaps the easiest way for this transfer to happen is for the client to go and get -- or **pull** -- the thing from the server. We do this all the time. For example, this is what happens when we go to the grocery store.

- The client requests the service as it is needed.
- The server handles each request as it comes in.

The following icon represents a client pull client: In this icon, information flows from right to left. The client (the circle) initiates the transfer of information, requesting it from the server and retrieving it. Here is a client pulling from a (passive) server:

Getter methods are very simple versions of client pull. In a getter method, one entity asks another for something; the method return completes the pull request. When direct method invocation is not available -- as when communicating over a channel, or network -- a pull request usually consists of two separate messages: one from the client, requesting the service, and one from the server, completing it.

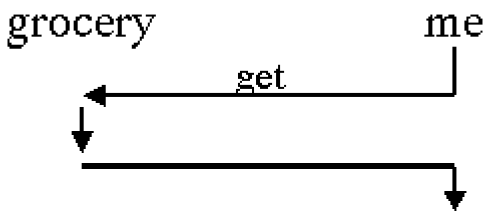


Figure #. A client pull interaction: I get what I need from the grocery store. Vertical lines represent computations that happen within an entity. Horizontal lines represent communication between entities. When I ask the grocery store for something, the grocery store gives it to me.

Locating the Server

In order for a client pull interaction to work, the client must know where to find the server. This can be accomplished either by telling the client about the server when the client is created or by providing a standard place to look. For example, I might first ask the phonebook where the grocery store is, then get what I need from the grocery store. This interaction involves two separate client pulls and is depicted in the next figure. Structurally, this is the way that computers locate each other on the internet.

Figure #. First I (client pull) get the grocery store's location from the phone book, then I (client pull) get the food I need from the grocery store.

Client Pull Tradeoffs

There are many advantages to client pull. The server doesn't have to do anything unless a request is pending. The client gets only what it needs, when it needs it. The client exercises control over the interaction, so the interaction (theoretically) happens when and where the client is best able to make use of the service.

On the other hand, there are disadvantages, too.

- The burden is on the client. (You have to go to the grocery store. Sometimes, you may miss out on a special because you get to the store at the wrong time.)
- The server may be deluged with requests and unable to keep up. (The grocery store may run out of something.)
- The network may be full of requests, since each client is sending these requests separately. (The check-out line may be long.)
- In general, more energy will be expended. For example, there's likely to be a lot more (network) traffic. (Each client arrives in his/her own car. Sometimes there are traffic jams in the parking lot.)
- The server's load may be patchy and unpredictable -- too busy one moment, unused the next -- making inefficient use of the machine and its resources. (Adequate inventory and staffing levels may be hard to identify, making the grocery store an inefficient business.)

Client pull works well when client requests are highly variable but overall not too great a load on the server. It allows each client to do its grocery shopping precisely when it needs to. When it doesn't work, though, the grocery store can be quite a mess!

Server push

An alternate architecture that addresses some of these problems is for the server to take initiative. In this case, it can simply deliver -- or **push** -- the information to the client when it is ready. This is sort-of like the fruit-of-the-month club. Every month, the fruit-of-the-month club delivers a box of fresh, ripe fruit to your house.

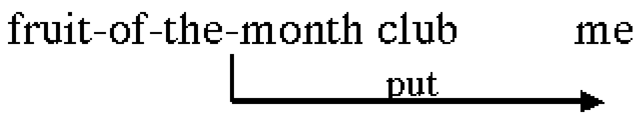


Figure #. The fruit-of-the-month club delivers a box of fruit to me each month, without my having to do anything. This is server push.

In direct method invocation, setter methods are server pushes. Although these methods technically complete with a return, no value is returned; in a put, only one-way communication is necessary. In channel- or network-based interactions, a push is often implemented as a single communication.

We can represent a server push server with the icon and a server push interaction with a (passive) client with the icon . Again, information flows from left to right. In this case, however, the entity with the information initiates the service.

Registering with the Server

Before the fruit-of-the-month club can provide me with regular deliveries, however, I may need to register my interest with the club. This is often done as a one-shot communication that precedes the regular (server push) delivery. Many subscription services -- like the fruit-of-the-month club, magazine subscriptions, or other periodic deliveries -- require a registration before the recurrent server push can begin.

Server Push Tradeoffs

There are numerous advantages to server push approaches:

- The server gets to control who gets what when. This means that it can manage its resources and keep its load even (or at least predictable): clients with names A-G this week, H-M next; oranges this month because they're in good supply.
- If the server is supplying multiple clients with the same -- or similar -- information, there may be economies of scale. For example, the server may only have to package up the information once to send to multiple clients.
- The client doesn't have to do anything to make this happen. The service just shows up whenever the server thinks it appropriate.

But this model doesn't always work perfectly, either. Why not? Let's consider what can go wrong:

- Deliveries might come at a bad time. Imagine that a whole month's shipment of fruit arrives the day after you've left town for a week. By the time you get back, the fruit will have spoiled. This happens in computational systems, for example, when the server goes too fast for the client, and the client has to spend all of its time handling the server's shipments. (Sometimes, this happens even if all that the client is doing is throwing the server's information away: trash can pile up and become overwhelming. In other cases, the client *can't* throw the information away, because the next delivery depends on the previous one in some crucial way.)
- Even though the server maintains control, it can still get too busy and deliveries can get back-logged. Some clients might need more frequent attention. Such a client might not get what it needs often enough, or even in time. The fruit-of-the-month club might be reliable, but not all computational servers are.
- Sometimes, the server doesn't know that (or when) the client wants or needs something.

Both the pros and cons of this approach can be summed up by the following:

- The client has very little control over what it gets when.

- The server has lots of control, but also has to do all of the work.

One popular way of doing animation in the early days of the web involved having the web server regularly push the next image in the animation sequence. This often swamped clients -- web browsers and the machines running them -- making it difficult for their users to do anything at all and giving server push a(n undeservedly) bad name.

The Nature of Duals

Server push and client pull are opposites of a special sort. The positive aspects of one are the negative aspects of the other. In general, they are like mirror images. Pairs of opposites like this are called **duals**, and they have some special properties. For example, you can generally take almost any statement expressed in terms of these dual operations (and their associated dual terms, such as client and server) and replace each operation with its dual without changing the truth or falsity of the statement:

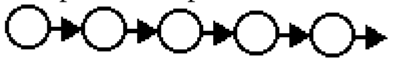
- Client pull gives the client a lot of flexibility, but the server doesn't have much control over its workload.

dual statement:

- Server push gives the server a lot of flexibility, but the client doesn't have much control over its workload.

Of course, it's not quite this simple -- it's not *too* hard to find statements that you can't turn around this way -- but client pull and server push *are* duals, which mean's that there's a fundamental symmetry in the ways that they work.

Pushing and Pulling Together

It is possible to build a system that uses multiple -- chained -- server pushes to produce its result. In this case, the client of one push becomes the server of another push:  For example, a farmer may push produce to the wholesaler -- taking it to market when it is ready -- while the wholesaler in turn may deliver it to the retailer when it becomes available.

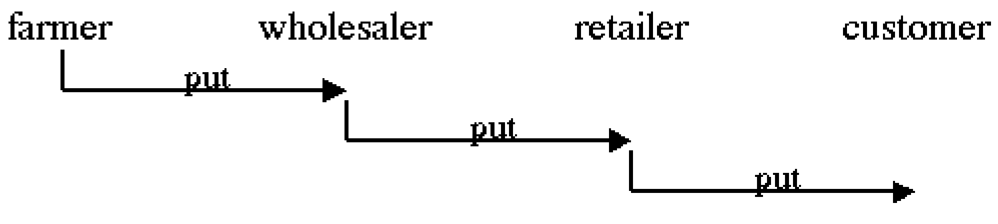


Figure #. You can chain together server pushes: the farmer sells to the wholesaler, who sells to the retailer, who sells to the customer.

Similarly, a chain of client pulls -- requests for services -- allows one client to pull from a server that may in turn request assistance from another service: Requesting a book on interlibrary loan follows this process.

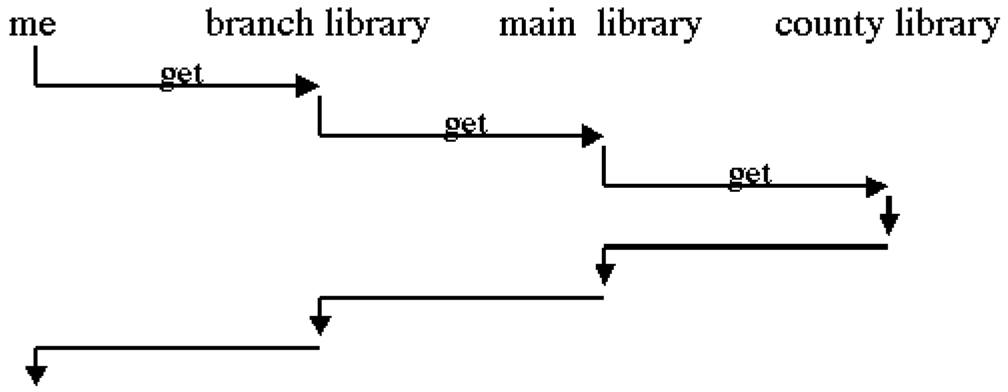


Figure #. You can also chain together client pulls: I reserve a book at my branch library, which asks the main library, which sends out the request to the entire county system, which finally finds the book.

Note, however that each of these pictures involve the chaining of similar service models. It is less simple to put a client pull client together with a server push server (or a server push client with a client pull server). (Iconically, there's no way to connect to or to .) To make these transactions possible, we need to introduce additional machinery. You cannot connect server pushes to client pulls (or client pull to server pushes -- there's that dual thing again!) without putting something different in the middle.

Passive repository

A passive repository is essentially just a the server side of a client pull combined with the client side of a server push. In other words, it's the passive recipient of information provided to it, and the passive provider of information when requested. It corresponds to the "drop box" where a spy might leave information for his spy master. The server -- or spy -- can drop off the information any time it wants. The client -- or spy master -- can come by and pick up the information whenever it is convenient. Iconically, this is just a . It has the important property that it can be used to connect a server push () with a client pull (), making a functioning system:

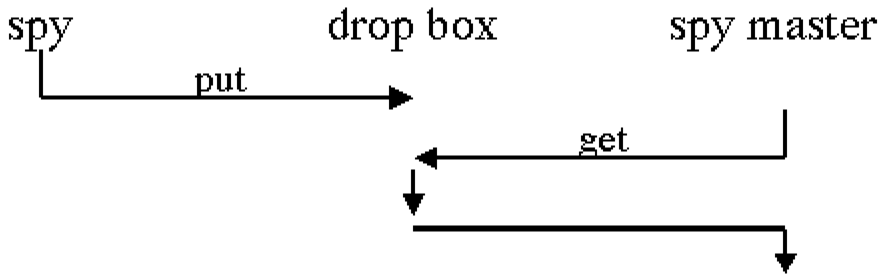


Figure #. A passive repository, like a spy's drop box, accepts both server pushes and client pulls.

What happens if the server pushes a new value before the client pulls the previous one? One possibility is that the passive repository actually contains a queue, i.e., keeps track of each of the items it's been given and provides them to the client pull client, either one at a time or all at once when the client requests them. (The first case, in which the repository provides the values one at a time, works much like an event queue -- see chapter 15 -- or channel -- see chapter 21.)

Alternately, the repository can keep track only of the last thing deposited. In this case, the repository will work well as long as the server updates the repository often enough that the client is assured of reading a relevant value. If the client doesn't check the repository often enough, though, the client runs the risk of missing some values.

A passive repository can be implemented using a single piece of shared data. For example, the server push may use a setter method, while the client pull uses a corresponding getter method. The data may be simple -- a single value or object -- or complex, capable of holding many values, like a queue.

Of course, there are both benefits and disadvantages to the use of a passive repository. Advantages include the flexibility to allow the active components to act whenever it may be convenient for them. But problems may arise:

- If the server pushes too much more often than the client pulls -- the spy drops off a lot of documents, but the spymaster doesn't claim them -- the repository can fill up.
- If server doesn't push often enough for the client -- or if the client pulls to frequently for the server -- the client may receive out-of-date (stale) information or no information at all. (The spymaster can't pick up documents faster than the spy delivers them.)

The use of a passive repository works best when the client and server need to operate relatively independently, but run at about the same rate.

Active constraint

An active constraint is the dual of a passive repository. If a passive repository couples a server push server and a client pull client, then an active constraint couples a server push client and a client pull server. Each of these is a passive component -- ○ -- so they must be joined by a component that takes action: . Note that this component both pulls (from the passive server) and pushes (to the passive client):

. Imagine a diner in a fancy restaurant. As soon as the diner puts down his fork, the fork disappears from the table, reappearing at the dishwasher. How does this happen? The active constraint -- in this case, the busboy -- gets (pulls) the fork from the diner and gives (pushes) it to the dishwasher.

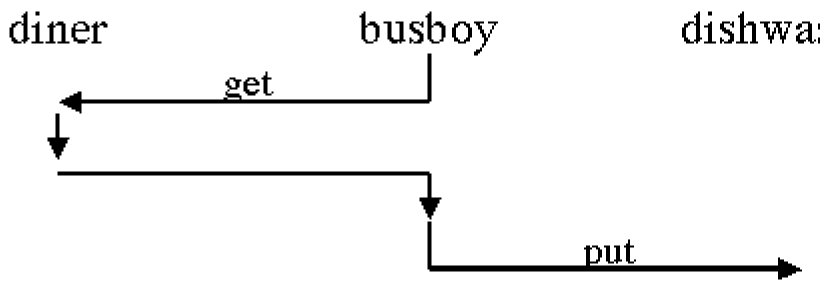


Figure #. An active constraint pulls information from the passive server and supplies it to the passive client.

This process requires no initiation of action on the part of either the client or the server. Instead, each of them goes about their business, responding only when the active constraint explicitly asks for something (or provides something). The intermediate entity -- the active constraint -- does all of the work to make this transfer happen.

Like server push and client pull, passive repository and active constraint are duals. A server push server can be connected to a client pull client by a passive repository. A client pull server (i.e., a passive server) can be connected to a server push (passive) client by an active constraint. In fact, a passive repository IS the client side of a server push attached to the server side of a client pull. By duals, an active constraint should be the server side of a server push and the client side of client pull -- and it is!

Chapter Summary

- A service is something provided by one entity to another. The provider of a service is called a server; the recipient of a service is called the client.
- An entity is a server or a client *with respect to a particular service.* Services can be layered or chained.
- Client pull describes the situation in which a client initiates a service request. This is like shopping at a grocery store, with all the attendant advantages and disadvantages.
- Server push describes the situation in which a server initiates the request. This is like subscribing to the fruit-of-the-month club.
- Client pull and server push are an example of duals.
- A server-push server and a client-pull client can be connected using a passive repository.
- A client-pull server and a server-push client can be connected using an active constraint.

Exercises

Real-world interactions are often complicated mixes of clients and servers. One way to tell who is (apparently) the server is that the client often pays for a server's services. Consider each of the following interactions and describe who is the client and who the server:

1. I buy a computer from a store.
2. I rent a car.
3. I rent a computer from a store.
4. I rent a computer from a service that (a) doesn't charge me but (b) requires that I read ads before using the computer.