**Protocols for communication**[Report Issue](https://github.com/LeetCode-Feedback/LeetCode-Feedback/issues)

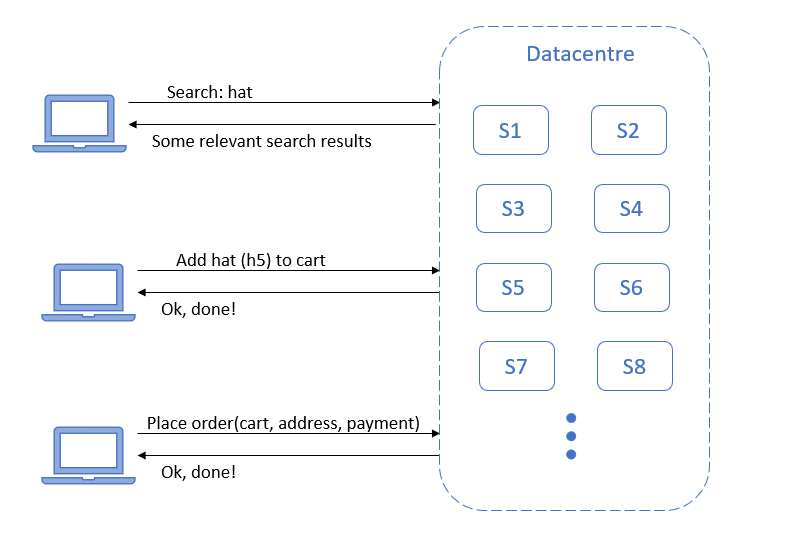
Some of the things that we'll cover in this module are:

1. What protocols are relevant to System Design
2. When should you use which protocol to talk to users and clients
3. What are the tradeoffs that you need to consider when making this decision

Please note, when we say users, we mean the users who are using the system and by clients we mean the devices the users are using to access the system.

How Clients and Servers Interact

Let us look at an example of a client-server interaction in the case of an e-commerce platform.



Say our user searches for a hat. Now the client will send a request to the server to search for hats and the server will respond with search results.

If a user wants to buy a hat, the client will send a request to the server to add the hat to the user's cart, and the server will respond with a confirmation.

If the user is happy with the items in their cart, on their action, the client will send the request to the server to place the order, and again, the server will respond with a confirmation.

In a real-world scenario, rather than talking to a specific server, the client’s request will instead be sent to a data center, where it could be picked up by any of the servers. However, irrespective of which server receives the request, the response will be the same. Based on this flow, we can draw the following conclusions about this architecture:

* It is client-driven. Only on the user’s button click will the client send the requests to the server, and the server will only respond to these requests.
* It is a simple request-response model. For every request from the client, the server will respond with some information or a simple confirmation.
* There are occasional requests from clients, only one request every few seconds based on the user's actions i.e. from the client-side it is a low throughput system.
* It is a stateless system i.e. irrespective of which server is responding, the response remains the same.

HTTP

These requirements make this a perfect use case for HTTP(s) protocol. Although these days, most architectures on HTTP have moved to HTTPS, which is a more secure version of HTTP as it prevents man-in-the-middle attacks.

Now, when we are using HTTP, REST is usually the best API standard to follow as it is very widely used and very user friendly.

Let us look at an example for a REST request and response:

Request:

Method: GET

URL: https://www.twitter.com/user/{id}

Response:

Status: 200 OK

Headers: <...>

Body: {

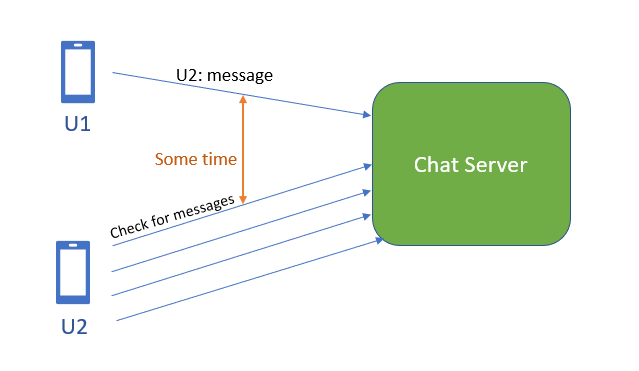
“userId”: 1,

“Email”: “someone@example.com”

}

The client makes a request to twitter.com over HTTPS to get information about a user with an id. In response, the server sends a success status code along with the user’s user id and email. As you can see, REST API standard is pretty much self-documenting, which adds to its user friendliness.

Now let us look at an example of a chat application.



We know that HTTP is a client-driven protocol, so the server cannot initiate any contact with the client. It can only respond to the client upon receiving a request. So when U1 sends a message to U2 via chat server, U2 doesn’t receive the message until it asks the server to share any pending messages. This leads to a delay when receiving messages.

A solution to this would be that U2 sends frequent requests to the chat server in the hopes of receiving a message. But this puts a huge load on the chat server as it will receive a huge number of requests from all its clients.

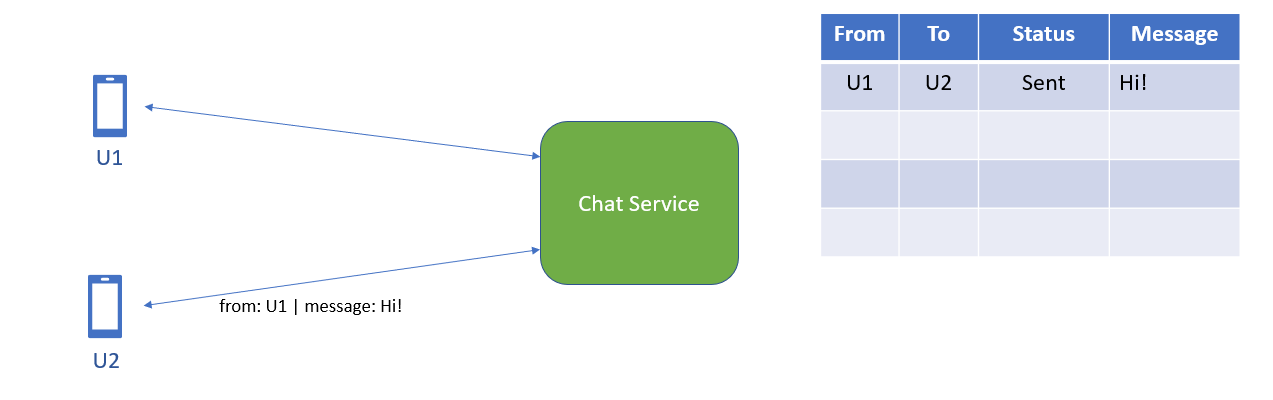
Another solution would be long polling. The server could wait for a few minutes before responding to U2, thus increasing the chance of a message being received before responding. But this is also not a very good solution.

The best approach would be if the server could send a notification to the user every time there is a message. For this, we use a protocol called WebSocket.

WebSocket

A WebSocket connection is a persistent connection. It is also a bidirectional protocol, where communication can be initiated by the client or the server as long as there is an open connection. It is optimized for high-frequency communication.

Let's look at how our chat application would work in the case of WebSocket protocol.



First, U1 and U2 will establish HTTP connections with the chat server, which are then upgraded to a WebSocket connection. When U1 sends a message for U2 via the chat server, it will store the message along with its status, RECEIVED, let's say.

The chat server, if it has an open connection with U2, will then send the message to U2 and update the status to SENT. If U2 was not online and there was no open connection between U2 and the server, the messages will be saved until U2 comes online and requests the server to send all pending messages. The server will send all messages with the status RECEIVED and update the status to SENT.

As you can see, with this approach we have:

* Reduced the latency, since the server can simply send the messages over an open connection
* Saved on CPU and bandwidth, as the client doesn’t need to unnecessarily send requests to the server and the server is not under unnecessary load
* Provided better user experience

Even with the benefits, there is a high cost to using WebSockets; that is the cost of maintaining a persistent connection with millions of users.

So how do we decide whether to use HTTP or WebSocket? Do we always go for Websocket then? Well, not really, as WebSocket is much more expensive than HTTP. We can safely say, if the communication between client and server is at a lower throughput on the client-side, HTTP is the way to go. If the communication is always client-driven, WebSocket is not needed. Also, if you are on a tight budget, HTTP may be the better choice.

On the other hand, if the communication from the client is at a higher throughput, WebSocket may be a better option. If the communication can be driven by both client and server, WebSocket is the way to go. Although here comes the tradeoff between cost and performance. We must decide if the optimization is really worth the huge cost of maintaining persistent connections with so many users.