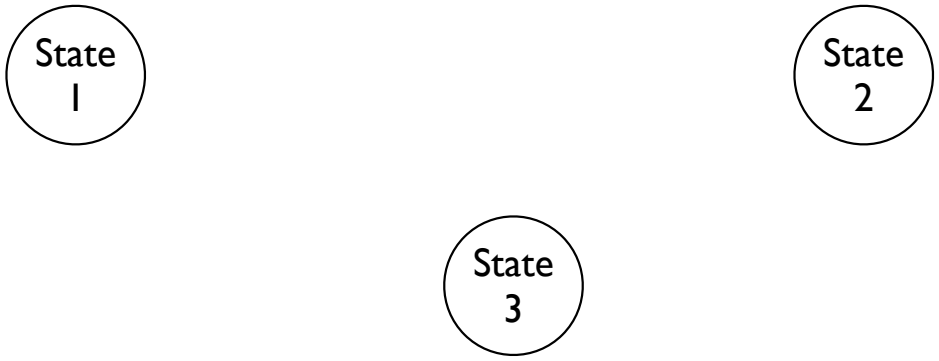
Finite State Machines

Protocol Specification

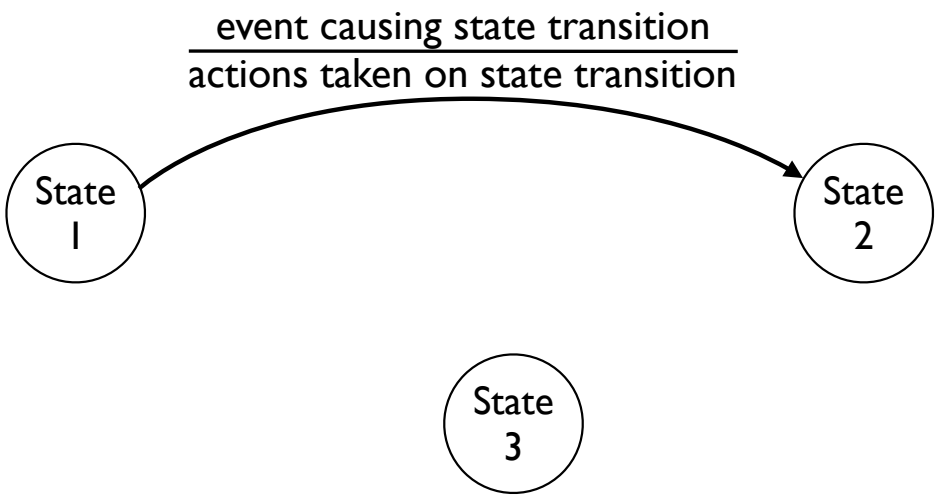
I'm going to explain finite state machines, something very commonly used when specifying network protocols and systems. I'll also explain the common way they're drawn in network protocols. I'll conclude by showing you the finite state machine that's part of the TCP specification, which defines how TCP connections are set up and torn down. So you'll see how you can describe something like the three-way-handshake of TCP in a finite state machine.

Finite State Machines



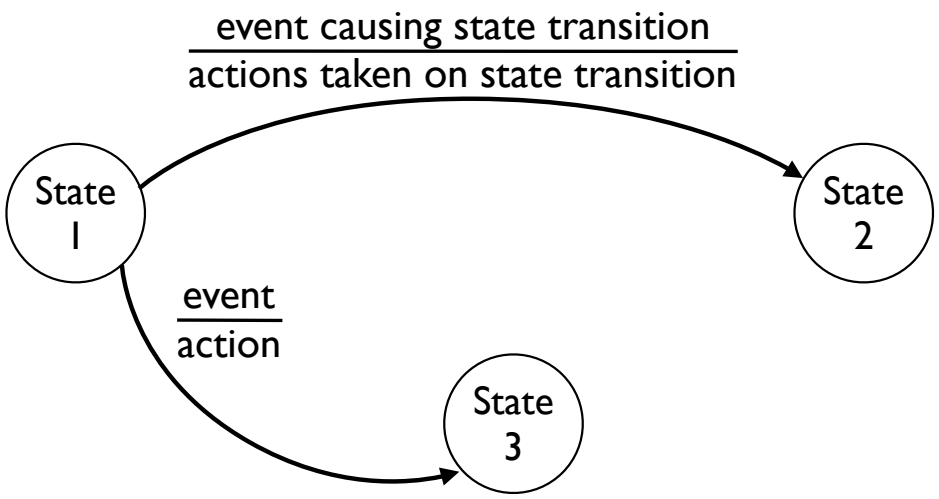
As the same suggests, a finite state machine is composed of a finite number of states. A state is a particular configuration of the system. I'm going to start with an abstract example. In this example, we have three states: state one, state two, and state three. So our system can be in one of these three states.

Finite State Machines



Edges between the states define how we transition between them. When we draw an edge, we first specify what events cause the transition to occur. Below this we can state what actions the system will take when that transition occurs. This part is optional, because not all transitions have actions associated with them. But if there is an action, you should specify it. Otherwise you have an incomplete specification and people might not test or implement it correctly. If the system is in a state and an event arrives for which there is no transition described, then the behavior of the FSM is undefined.

Finite State Machines



There can be multiple transitions from a single state. So here we have a second transition from state 1, a different event that will take the system into state 3. For any given state, the transition for an event must be unique. In this example, an event can cause state 1 to transition to state 2, OR transition to state 3. But you can't have the same event associated with both transitions, otherwise the transition is ambiguous. If the event occurs, are you in state 2 or state 3? The system can only be in one state.

FSM Example: HTTP Request

So let's walk through an example, an HTTP request. In practice HTTP requests are a bit more complex than this, there all kinds of options, so for this example we'll just use a very simple form.

Let's describe our system this way. In our starting state we are are viewing a page or otherwise idle. When we want to load a new page, we transition to the page requesting state. So the event is load new page, and the action is open a connection to the web server. Once we've opened a connection, we're now in the page requesting state. We'll transition back to the idle state when the connection closes or when we finish requesting every resource on the page.

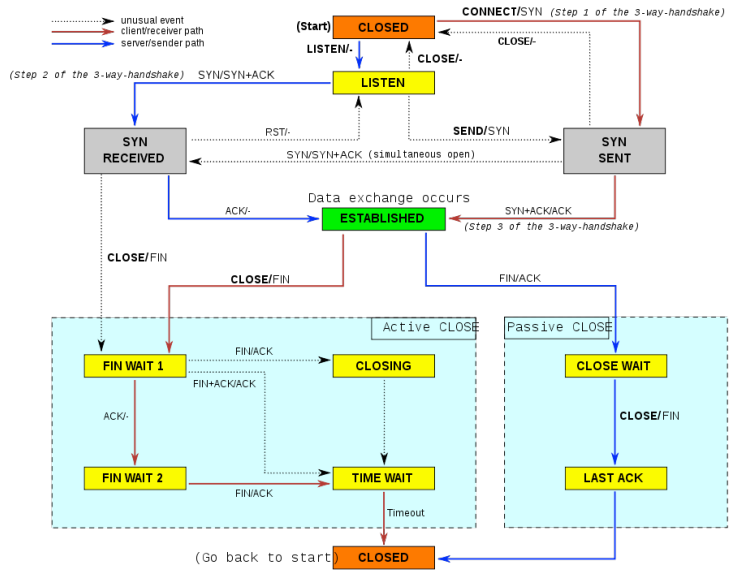
We need one more state, which describes where we are in requesting a page. On the event of having more resources to request, we take the action of requesting a resource with an HTTP GET. This puts us in the request pending state. On the event of receiving the response, our system transitions back to the page requesting state.

So here we have a three state system. Idle, page requesting, and request pending.

On one hand, this is a nice, simple FSM. But if you were to try to implement it, it leaves a lot unsaid. Specifically, we have 4 events in the system: page request, more requests, receive response, and connection closed. So what happens if the connection closes when we're in the request pending state? Or when we receive a page request while in the page requesting state? Or receive response while in the idle state?

If you want to be completely explicit and careful, you should specify what happens on each state for every event. But this can lead to complicated FSMs which have tons of edges. So often instead you'll write down just the common cases, for ease of understanding, and have some supporting text about other transitions. Or, in some cases, it can even be acceptable to leave something undefined. The Internet Engineering Task Force, for example, often doesn't completely specify every FSM. The idea is that by specifying only the parts that are necessary for interoperability, you can leave the specification flexible for future exploration. As people use the protocol, they'll figure out if something is important and if so can specify that extra part later.

FSM Example: TCP Connection



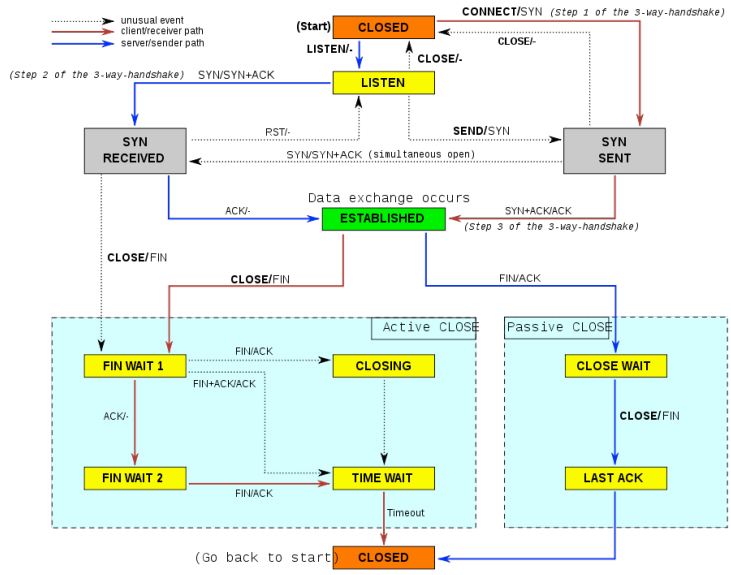
http://upload.wikimedia.org/wikipedia/commons/thumb/a/a2/Tcp_state_diagram_fixed.svg/

So let's walk through a real example of an FSM, probably the most famous FSM in the Internet. This diagram here describes the finite state machine of TCP. I know it looks very complicated -- it has 12 states -- but I'll walk though it bit by bit and you'll see how it all fits together.

First off, the diagram really has four parts, which we can look at separately. These top 4 states are what describe how you open a TCP connection. This center state, "ESTABLISHED" is when TCP is sending and receiving data. It's after the connection has been established but before it's been closed. These 6 states describe how connections close. This state at the bottom, CLOSED, denotes the connection has closed and the node can forget about it. Note that the top state is also the closed state -- before we open the connection.

Recall that you start a TCP connection with a three way handshake -- SYN, SYN/ACK, ACK. The client, or active opener, sends a SYN, synchronization, message to a program listening for connection requests. When it receives a SYN, it responds with a SYN/ACK, synchronizing and acknowledging the original synchronization. The active opener, on receiving the SYN/ACK, responds with an acknowledgement.

FSM Example: TCP Connection



http://upload.wikimedia.org/wikipedia/commons/thumb/a/a2/Tcp_state_diagram_fixed.svg/

The state diagram here describes how TCP behaves on both sides of the TCP three-way handshake. A passive opener is a server. It listens for requests for connections from active openers, clients. So when a program calls listen(), the socket transitions from the orange closed state to the yellow listen state. The protocol takes no actions when this happens -- it doesn't send any messages. If the server calls close on the socket when it's in the listen state, it transitions immediately to the closed state.

Let's walk through the three way handshake starting with the first step, when a client tries to open a connection and sends a SYN packet to the server. We can see this first transition for the client side of the connection as this orange arrow from closed to the SYN SENT state. This happens when the client program calls connect -- the event -- and the client sends a SYN message.

So once the first SYN is sent, the client is in the SYN SENT state and the server is in the LISTEN state. When the SYN arrives at the server, this leads to this blue transition. You can see the event is receiving a SYN message. The action is to send a SYN/ACK message in response. Now the server is in the SYN RECEIVED state.

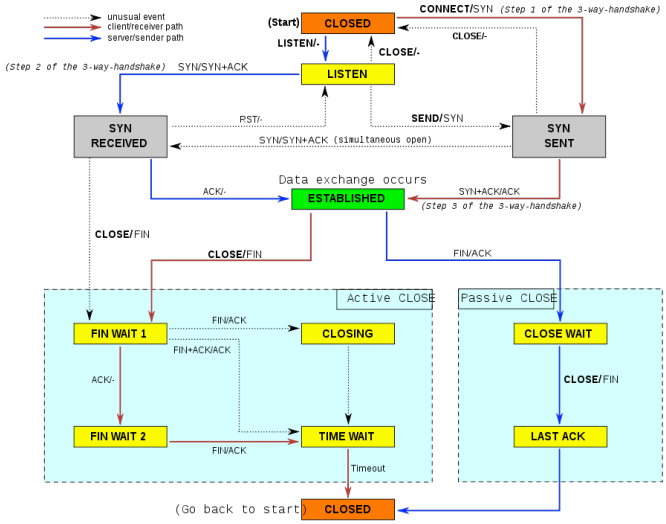
Let's jump back to the client. Remember, it was in the SYN SENT stage. Now, when it receives the SYN/ACK from the server, it transitions to the ESTABLISHED state. Its action is to send an ACK message, the third message of the SYN, SYN/ACK, ACK handshake. Now the client can start sending data to the server.

Finally, let's go back to the server, which is in the SYN RECEIVED state. When it receives the ACK from the client, it transitions to the ESTABLISHED state and can send data.

There are a couple more transitions during connection opening -- don't worry about them for now, I discuss them in detail in the video on connection setup and teardown.

Quiz

Assume there is no other documentation of the TCP finite state machine, so there's no supporting textual description which defines other state transitions.



Suppose we start in the closed state, then call listen, then receive a SYN, then call close. What state will we be in: CLOSED, SYN SENT, SYN RECEIVED, ESTABLISHED, FIN WAIT 1, or undefined?

Suppose we start in the closed state, then call connect, then call close. What state will we be in: CLOSED, SYN SENT, SYN RECEIVED, ESTABLISHED, FIN WAIT 1, or undefined?

http://upload.wikimedia.org/wikipedia/commons/thumb/a/a2/Tcp_state_diagram_fixed.svg/

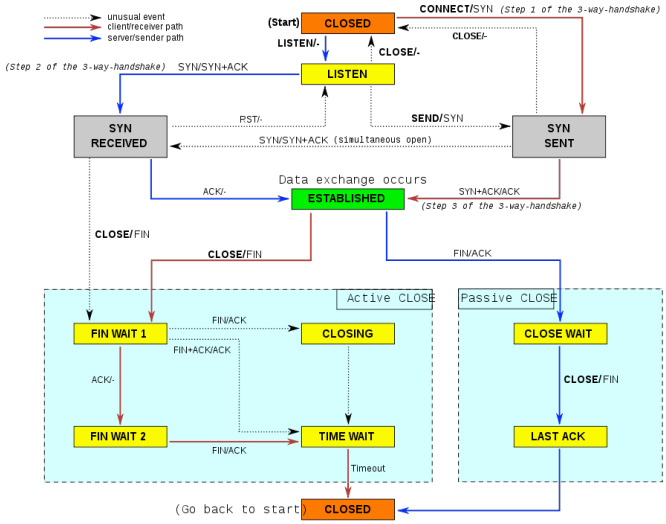
Here's a quiz. For this quiz, assume there is no other documentation of the TCP finite state machine, so there's no supporting textual description which defines other state transitions.

In the first question, suppose the finite state machine starts in the closed state. Then a user program calls listen on the socket. The socket receives a SYN message, but before any other event arrives the user program calls closed. What state will the socket be in?

In the second question, suppose the finite state machine starts in the closed state. Then a user program calls connect and before any other event arrives the user program calls close. What state will the socket be in?

Quiz

Assume there is no other documentation of the TCP finite state machine, so there's no supporting textual description which defines other state transitions.



Suppose we start in the closed state, then call listen, then receive a SYN, then call close. What state will we be in: CLOSED, SYN SENT, SYN RECEIVED, ESTABLISHED, FIN WAIT 1, or undefined?

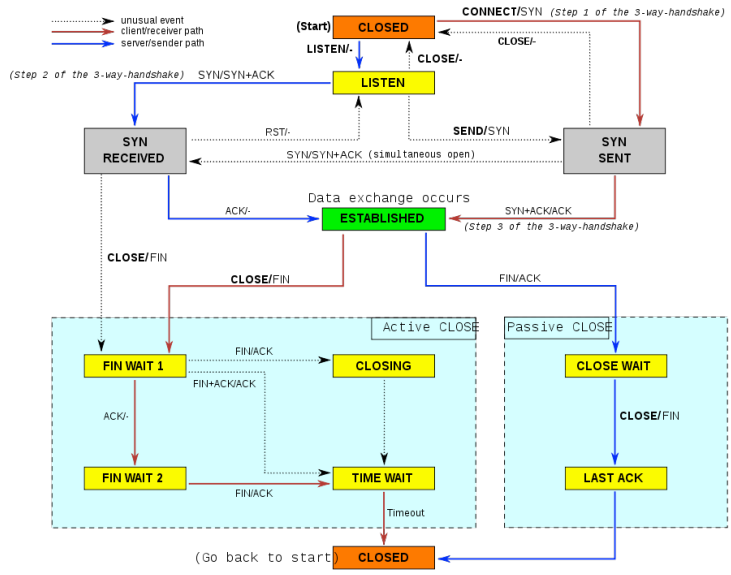
Suppose we start in the closed state, then call connect, then call close. What state will we be in: CLOSED, SYN SENT, SYN RECEIVED, ESTABLISHED, FIN WAIT 1, or undefined?

http://upload.wikimedia.org/wikipedia/commons/thumb/a/a2/Tcp_state_diagram_fixed.svg/

The answer to the first question is FIN WAIT 1. Let's walk through why. We start in the closed state. Then the user program calls listen. We transition to the LISTEN state. Then the socket receives a SYN and we transition to the SYN RECEIVED state. While in the SYN RECEIVED state the user program calls close. So we traverse the edge with close() as an event, to the FIN WAIT 1 state.

The answer to the second question is CLOSED. We start in the closed state. Then the user program calls connect and we transition to the SYN SENT state. While in the SYN SENT state the user program calls close. There's an edge from SYN SENT on the close event, back to the CLOSED state.

FSM Example: TCP Connection



http://upload.wikimedia.org/wikipedia/commons/thumb/a/a2/Tcp_state_diagram_fixed.svg/

So now our sockets are in the ESTABLISHED state. They're exchanging data. The six states in blue boxes are how TCP "tears down" a connection, or how it closes it. It's sometimes useful to talk about "tearing down" a connection because the word "close" means something in terms of system calls. A connection exists after one side "closes" it, as we'll see.

There's symmetry between how TCP sets up a connection and how it tears it down. Where connection establishment uses synchronization or SYN packets, connection teardown uses finish, or FIN packets. If one of the sides of the connection calls close, it traverses along the right edge on the left to the FIN WAIT 1 state. This causes it to send a FIN packet to the other side of the connection. This is called the "active closer" because it starts the operation. The other side receives the FIN and takes the blue edge on the right to the CLOSE WAIT state. It remains in this state until the program on its side calls close, at which point it sends a FIN.

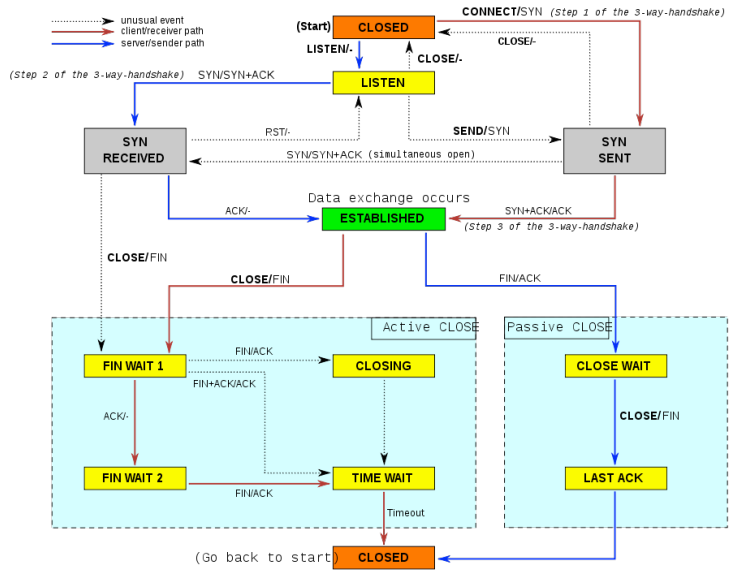
Here's where it gets a little complicated. A TCP connection is bidirectional. The active closer has closed its direction of the connection, so it can't write any more data. But it could be the passive closer has more data to send. So it can continue to send data, which the active closer receives and acknowledges. Or it could close its side of the connection to. Or it could even have decided to close the connection at the same time, such that we have two FIN packets crossing each other in the network.

From the FIN WAIT 1 state, where the active closer is, there are three possible outcomes. First, the passive closer might acknowledge the FIN but not send a FIN. In this case, the passive closer is in the CLOSE WAIT state and can continue to send data. This is the lowermost edge, where the active closer enters the FIN WAIT 2 state. Second, the passive closer might close its side too, acknowledging the FIN and send a FIN of its own. This is the middle edge, to the TIME_WAIT state. Finally, it could be that both sides actively closed at almost the same time, and sent FINs to each other. In this case, both are in the FIN WAIT 1 state. Each one will see a FIN from the other side that doesn't ACK its own FIN. In this case we transition to the CLOSING state, and when our FIN is acknowledged we transition to the TIME_WAIT state, just as with the middle edge.

TCP transitions from FIN WAIT 2 to TIME WAIT when we receive a FIN from the other side. It then stays in TIME WAIT for a period of time, until it can safely transition to close.

The final blue edge, from LAST ACK to CLOSED, occurs when the passive closer's FIN is acknowledged.

FSM Example: TCP Connection



http://upload.wikimedia.org/wikipedia/commons/thumb/a/a2/Tcp_state_diagram_fixed.svg/

On one hand, that's a lot of detail. There are 12 states, covering lots of cases. But you can see how this FSM takes what was previously a few colloquial descriptions and gives them detail and precision. Trying to implement a properly interoperating TCP based on those descriptions would be hard. This diagram precisely specifies how TCP behaves and so is tremendously useful.