TCP Connection Release

In this lesson, we'll discuss how TCP terminates established connections.

We'll cover the following

TCP, like most connection-oriented transport protocols, supports two types of connection releases:

1. **Graceful** connection release, where the connection is not closed until both parties have closed their sides of the connection.
2. **Abrupt** connection release, where either one user closes both directions of data transfer or one TCP entity is forced to close the connection.

Abrupt Connection Release

We've already had a brief overview of abrupt connection release with *RST* segments in a previous lesson. Let's have a closer look.

An abrupt release is executed when a *RST* segment is sent. A *RST* can be sent for the following reasons:

- A non-*SYN* segment was received for a **non-existing TCP connection** (RFC 793).
- Some implementations send a *RST* segment when a segment with an

invalid header is received on an open connection (RFC 3360). This

causes the corresponding connection to be closed and has prevented attacks (RFC 4953).

- Some implementations send an *RST* segment when they need **to close an existing TCP connection** for any reason such as:
  - There are **not enough resources** to support this connection
  - The remote **host has stopped responding and is now unreachable**.

When a *RST* segment is sent by a TCP entity, it should contain the current value of the sequence number for the connection (or 0 if it does not belong to any existing connection), and the acknowledgment number should be set to the next expected in-sequence sequence number on this connection.

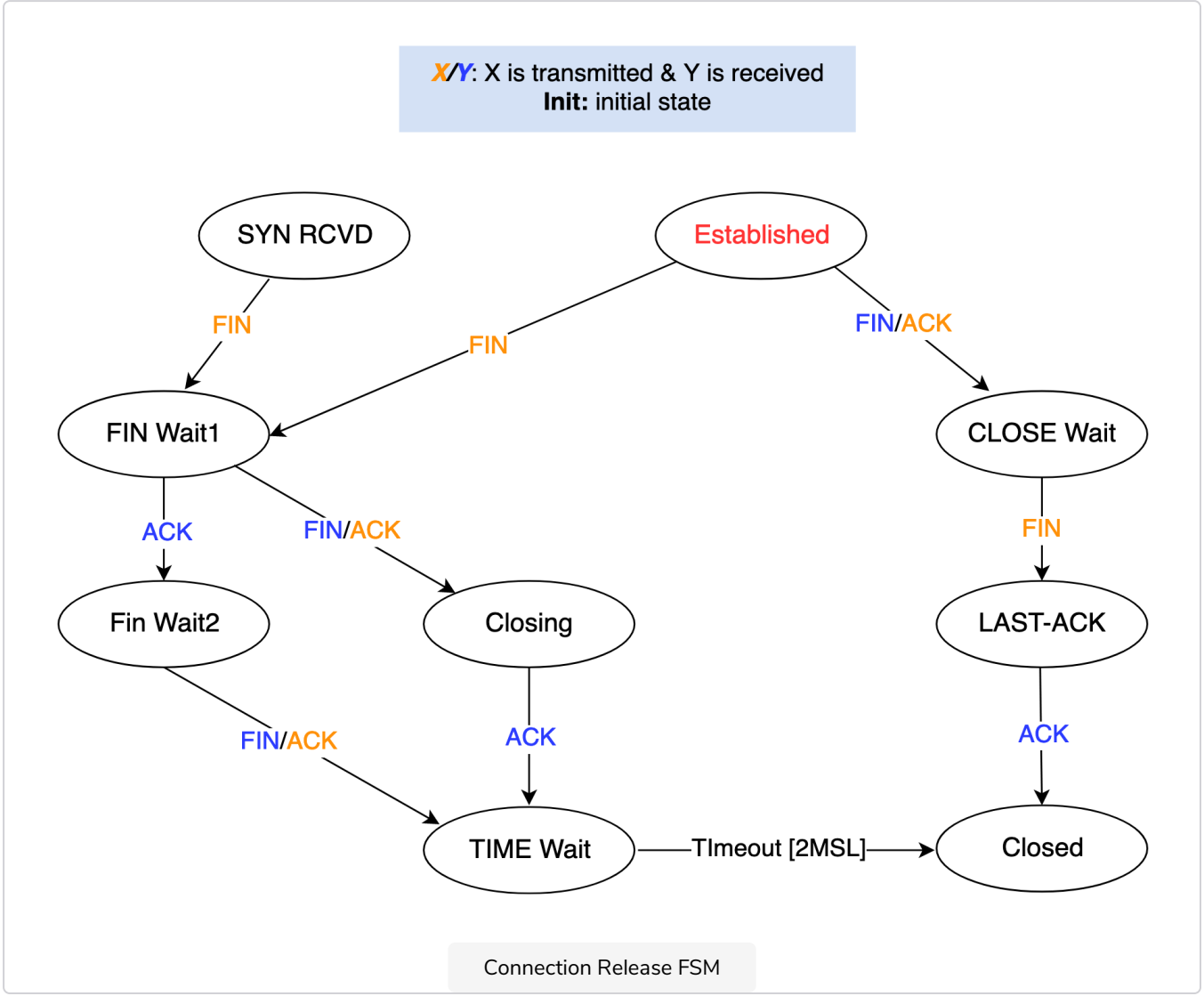
Graceful Connection Release

The normal way of terminating a TCP connection is by using the *FIN* flag of the TCP header. This 'graceful mechanism' allows each host to release its own side of the connection individually. The utilization of the *FIN* flag in the TCP header consumes one sequence number.

FSM

The following figure shows an FSM that depicts the various 'graceful' ways that a TCP connection can be released.

Don't feel overwhelmed if you don't understand it yet, we'll study each possible path individually.



Connection Release FSM

Starting from the **Established** state, there are two main paths through this FSM.

Receiving a FIN

Throughout the rest of this lesson we'll refer to the two hosts as **client** and **server**. In the case of this path, the client receives a *FIN* segment. Let's trace it.

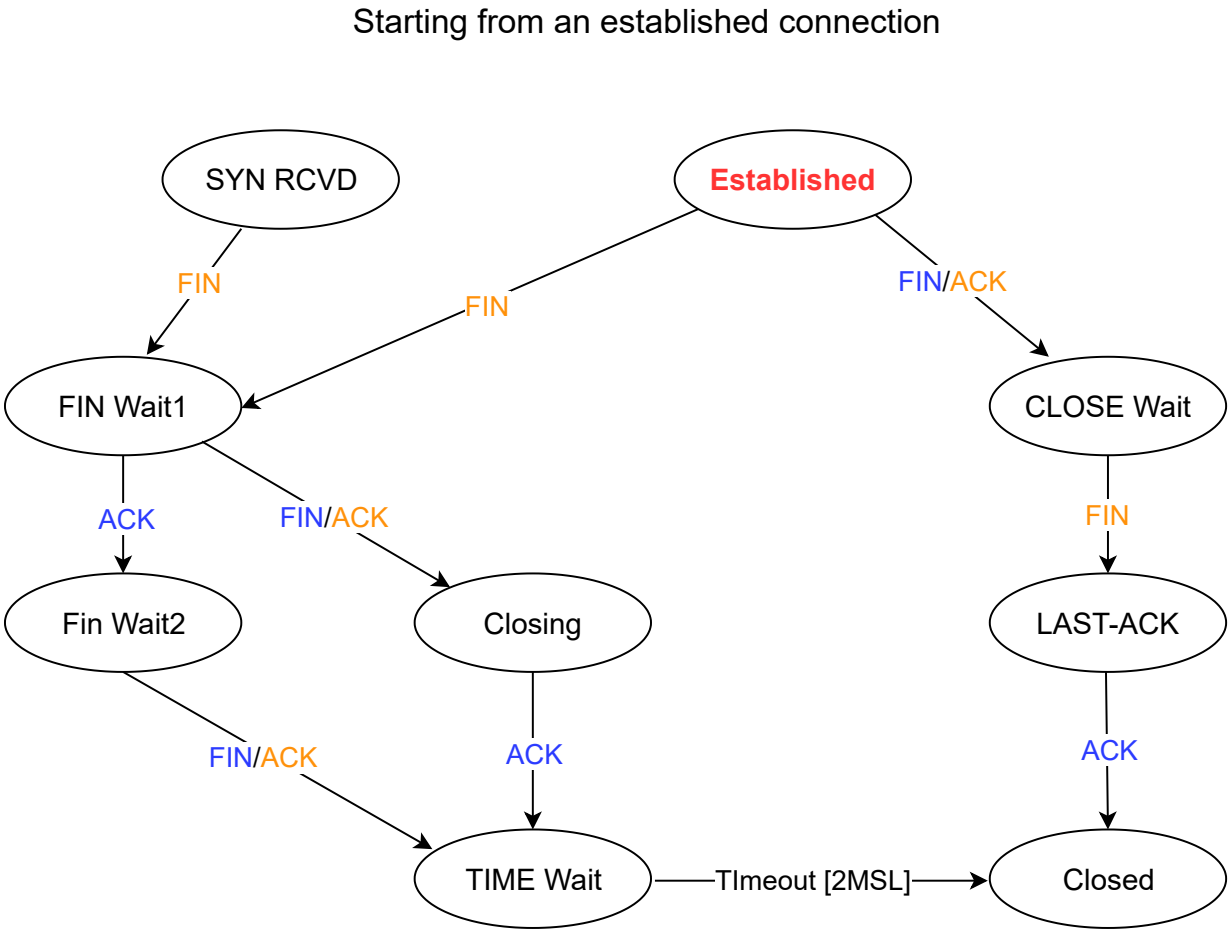
1. The client receives a segment with sequence number x and the *FIN* flag set. The utilization of the *FIN* flag indicates that the byte before sequence number x was the last byte of the byte stream sent by the server. The *FIN* segment is subject to the same retransmission mechanisms as a normal TCP segment. In particular, its transmission is protected by the retransmission timer that we'll look at in the next few lessons.
2. Once all of the data has been delivered to the application layer entity, the TCP entity sends an *ACK* segment to acknowledge the *FIN* segment it received in step (1), whose **acknowledgment number** field is set to

$(x + 1)\ mod\ 2^{32}$.

3. At this point, the TCP connection enters the **CLOSE_WAIT** state. In this state, the client can still send data to the server.
4. Once the client has sent all the data that it was supposed to, it sends a *FIN* segment and enters the **LAST_ACK** state. In this state, the client waits for the acknowledgment of its *FIN* segment. It may still retransmit unacknowledged data segments, e.g. if the retransmission timer expires.
5. Upon reception of the acknowledgment for the *FIN* segment, the TCP connection is completely closed and its TCB can be discarded.

Tracing through the FSM

Here are some slides tracing this path through the FSM.



Connection Release FSM

1 of 6

Sending a FIN

The second path is when the client decides first to send a *FIN* segment.

1. Upon sending the *FIN*, the connection enters the **FIN_WAIT1** state. In this state, the client can retransmit unacknowledged segments, but cannot send *new* data segments. There are two paths that this one can split into after this:

    - **A.** The client receives an ACK segment in response to its FIN. The TCP connection enters the **FIN_WAIT2** state in which new data segments from the server are still accepted until the reception of a *FIN* segment. The acknowledgment for this segment is sent once all the data before the *FIN* segment has been delivered to the client. After this, the connection enters the **TIME_WAIT** state.

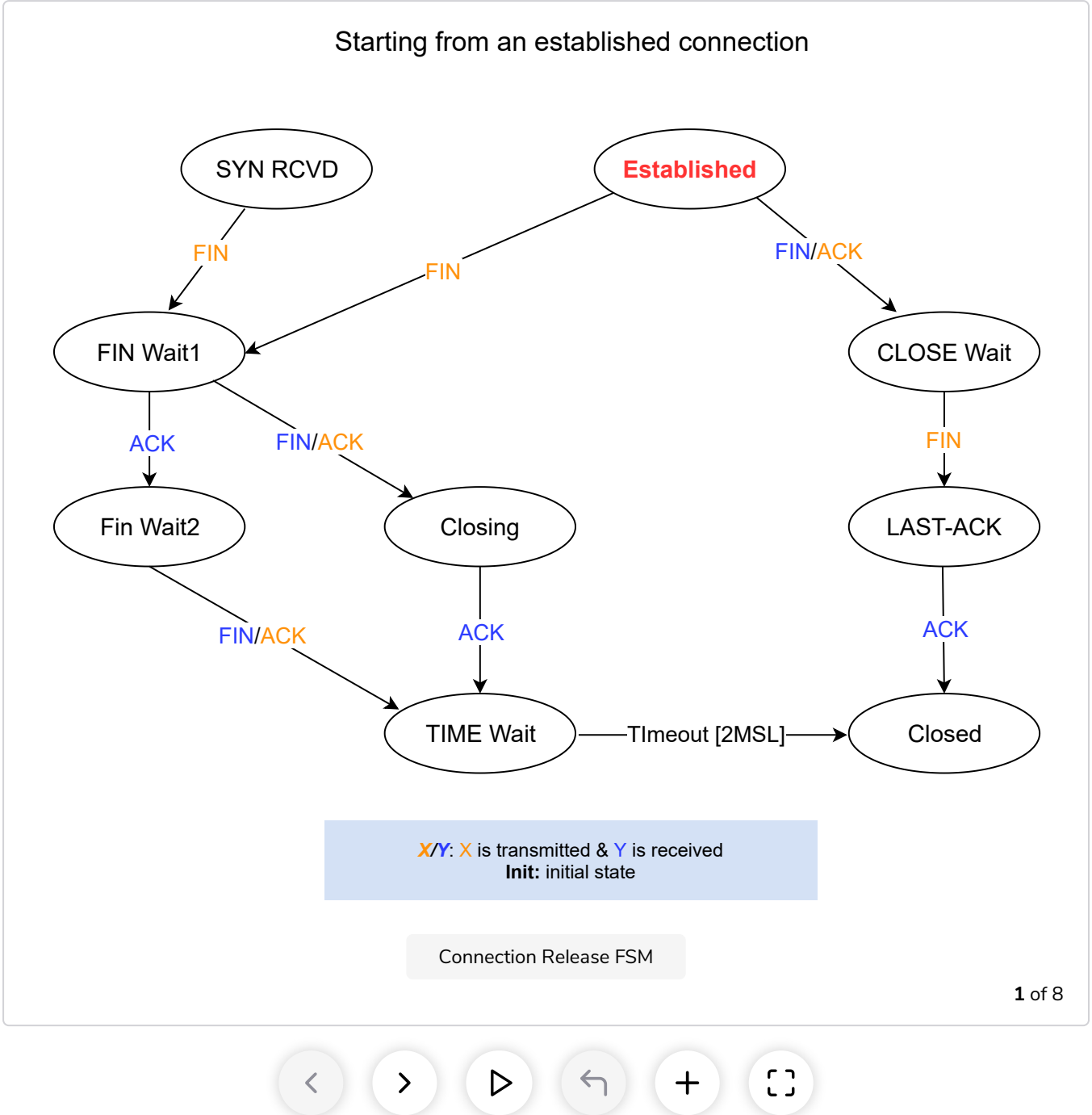
    - **B.** In the second case, a *FIN* segment is received from the server. The connection enters the **Closing** state once all the data from the server has been delivered to the client. In this state, no new data segments can be sent and the client waits for an acknowledgment of its *FIN* segment before entering the **TIME_WAIT** state.

2. A TCP connection enters the **TIME_WAIT** state after the client sends the last *ACK* segment to a server. This segment indicates to the server that all the data that it's sent has been correctly received and that it can safely release the TCP connection and discard the corresponding TCB.

3. The connection remains in the **TIME_WAIT** state for $2 \times$ a certain length of time called the **maximum segment lifetime** (MSL) seconds. The TCP standard defines MSL as 120 seconds (2 minutes). Although, this value is flexible in modern implementations. During the $2 \times$MSL period, the TCB of the connection is maintained on both ends to:

    - allow retransmission of the sent *ACK* segments if any are lost.
    - handle duplicate segments on the connection correctly without causing the transmission of a *RST* segment

Slides of Path 1A #



Quick Quiz!

1 What are some reasons why a connection may get abruptly terminated?

A) The server has sent all the data that was requested

B) The server does not have enough resources to sustain the

connection

C) The client has received all the data that it needed and the connection is now redundant

COMPLETED 0%

1 of 3

Now that we're done with what connection release looks like in TCP, let's move on to efficient data transmission with TCP using Nagle's algorithm in the next lesson!