



# HTTP/2 in Erlang

Joe DeVivo

 @joedevivo

 [github.com/joedevivo/chatterbox](https://github.com/joedevivo/chatterbox)

 [joedevivo.com](http://joedevivo.com)

 #http2erlang

Strange Loop 2015

I found Erlang and HTTP/2 to both be intimidating when I first started with them. Hopefully this talk will make them both more accessible.

# WHY HTTP/2?

2014's		
HTTP/1.1 Special Edition		
RFC-7230		
RFC-7231	HTTP/2	HPACK
RFC-7232	RFC-7540	RFC-7541
RFC-7233		
RFC-7234		
RFC-7235		

HTTP/1 will be around for a long time. This is good, since it defines a lot of semantics that are still valid in HTTP/2. Also, HTTP/2 servers should be able to upgrade HTTP/1 connections to HTTP/2 if both sides support it with the **101 Switching Protocols** response code.

HTTP was originally designed by physicists to share papers with “linked” references. But now we uses it for everything!

The actual 'Why?' is efficiency. HTTP was built for TEXT by physicists who wanted to share academic papers and thought it would be cool if they could "link" their references to the actual reference.

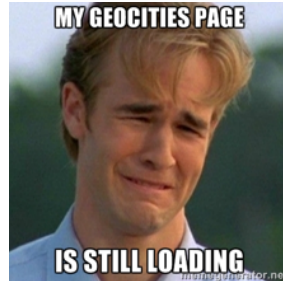


## 1 Introduction

Hi. I'm Joe DeVivo and I'm here to talk about HTTP/2 in Erlang. We're not going to get to it all. I cut tons from this talk. Hopefully I can give you all enough context to go learn more. So which should I talk about first?

### 1.1 Why HTTP/2?

HTTP/2 is the future. It's coming and there's no stopping it. I actually haven't even used HTTP/1 in months. Hahaha, just kidding. HTTP/1 will be around for a long time.



for TEXT by physicists who wanted to share academic papers and thought it was better to use a reference.

pages, javascripts, videos, you name it, it could be sent over HTTP. And it was in the mid 1990's and I had 90's web problems. If you were there too, you remember that thing where the page loaded in black and white, with the default font and background. Shout out to underconstruction.gif

It improved with HTTP/1.1, right?

As at solving things like this (e.g. Pipelining), this problem really got solved by perceived latency went down as the masses, myself included, got access to broadband.

Because slower connections are back in a big way with mobile devices, edge computing, and not to mention the "internet of things".

When talking about HTTP/2, remember that it's stingy. Why use two bits when one will do. That also means taking advantage of a persistent TCP connection, instead of

It changes how efficiently we use the wire without changing HTTP/1.1's semantics.

### 1.2 Why Erlang?

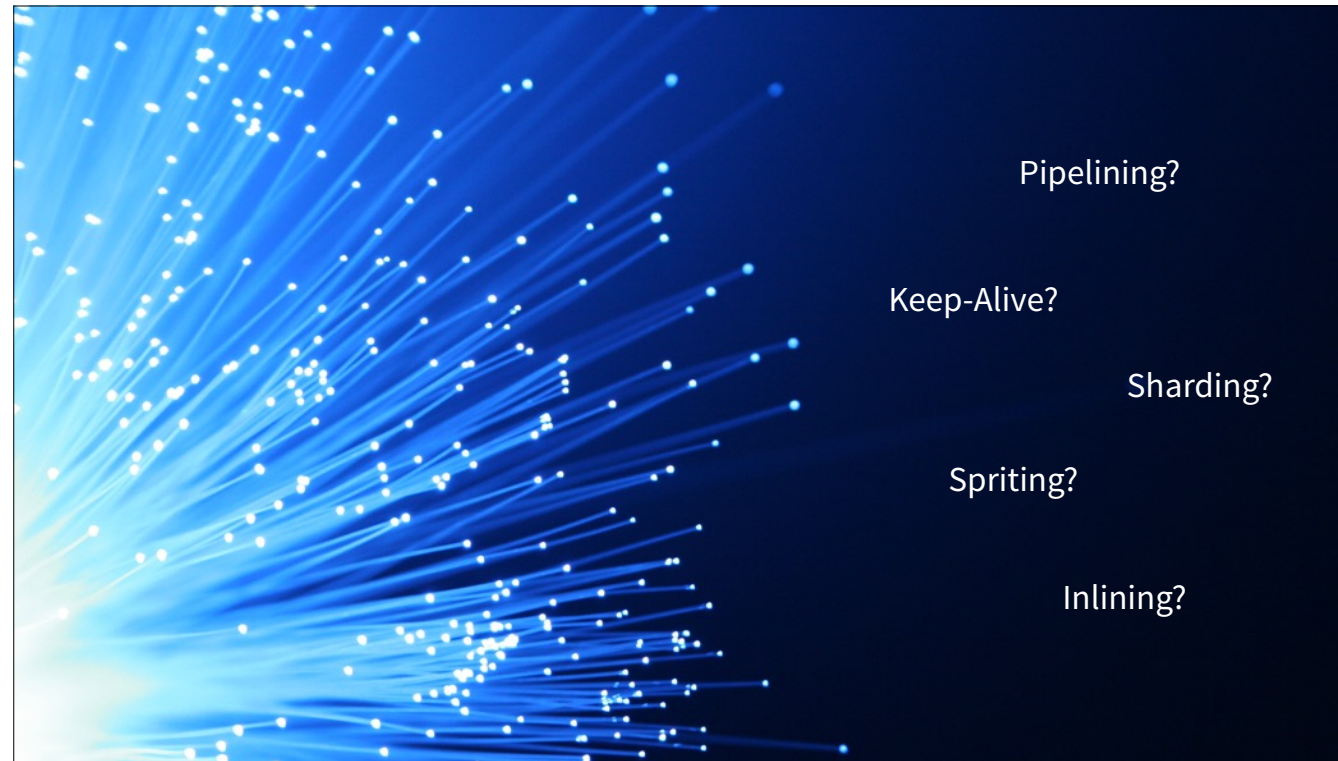
The really short answer is that I'm an Erlanger. I've been working with Erlang web technologies, and saw that Erlang would eventually have a need for an HTTP/2 server, (and client for that matter). So, "because I love Erlang" was reason enough for me, but you're not me. Are you?

In case you weren't sure; No, you're not. So let me talk a bit about why I like Erlang.

-----

I rolled up on the world wide web in the mid 1990's and I had 90's web problems. If you were there, you remember broken jpegs and unloaded style. We can't blame the protocol for underconstruction.gif, that was a design decision we all made together.

Did HTTP/1.1 fix this stuff?



HTTP/1.1 tried. Some parts (like Keep-Alive connections) succeeded, and others (like Pipelining) failed. Many parts of the HTTP/1.1 spec were optional, and I don't think that there's an HTTP/1.1 compliant server that implements **ALL** of 1.1's features.

So clever people came up with clever workarounds. Sharding, Spriting, Inlining all helped optimize a protocol that wasn't meant for this.

In the end, perceived latency went down as broadband adoption went up.

So good then, we all have broadband. Is that enough?





Nope! Slower connections are back in a big way!

- \* mobile devices,
- \* edge networks throughout the developing world
- \* “internet of things”

This is where HTTP/2 comes in



Bin?

More like **Binary**

HTTP/2 is like the Scrooge McDuck of protocols. Cheap cheap cheap!

The goal is to optimize the amount of bytes served over the wire. It uses a single persistent TCP connection. It's a binary protocol for the 1st time. People are mad about this, but we should all be using SSL now anyway, right?

If you only remember one thing from this talk, remember that **HTTP/2 changes efficiency without changing semantics.** hence, all the “new-old” RFCs are new again.



Why Erlang? Well, I love Erlang. But I also work at Chef, and Chef Server is written in Erlang. It generates lots of traffic, and I thought that one day CCRs could be served via HTTP/2.

When that day came, I didn't want to be held up by lack of existing server.  
I also wanted to know if I could do it?



Erlang rocks the actor model for concurrency.

Each actor

- \* lives in its own little world
- \* can receive messages from the outside, which it can then modify its world view based on.
- \* can send it's own messages out into the world
- \* can even create new actors.

In Erlang, the actors are processes, and it's extremely easy to spin up a new one.

# OTP

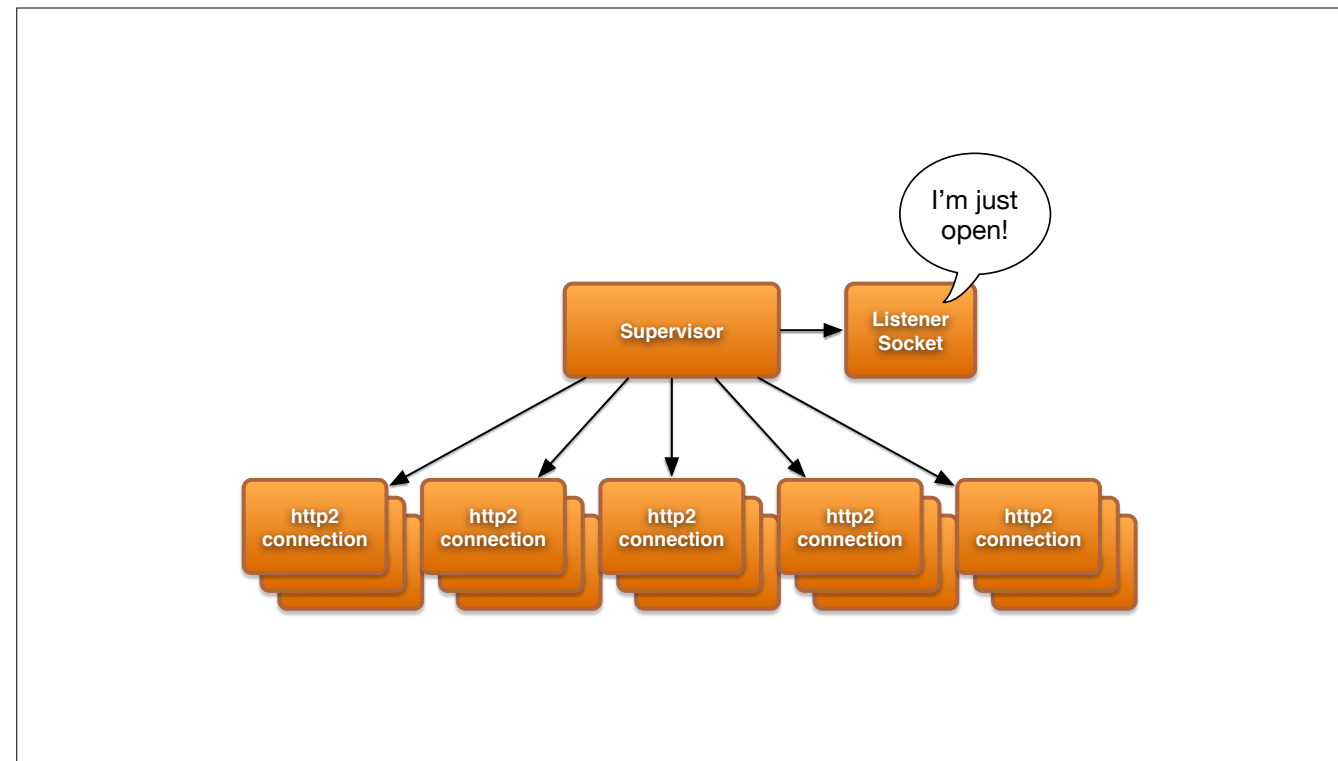
The Open Telecom Platform

Erlang comes with OTP, set of standard libraries that make working with the Actor model easier

# The Supervisor

Supervisors are all about process management.

I got [the one for chatterbox](#) almost verbatim from Learn You Some Erlang



What the supervisor is doing is setting up a socket to listen on whatever port we've configured.

- \* spawns N processes that wait to accept
- \* When a client connects, one process will handle

This is how Apache Prefork and Unicorn do it.

Erlang provides four supervision strategies, and for this one we've chosen **simple\_one\_for\_one**, which is like a factory pattern for actors. What's our factory make? http2 connections.

Simple\_one\_for\_one is also good at managing lots of children.

# The Generic Server

**gen\_server** to its friends

**NOPE**

So we're writing a server, might as well start with Erlang's Generic Server behavior. **gen\_server** is like a bring your own state. What it does for you is handle Erlang's process messaging API. It lets you write callbacks that answer the question "If someone asks for this, I'll do that"



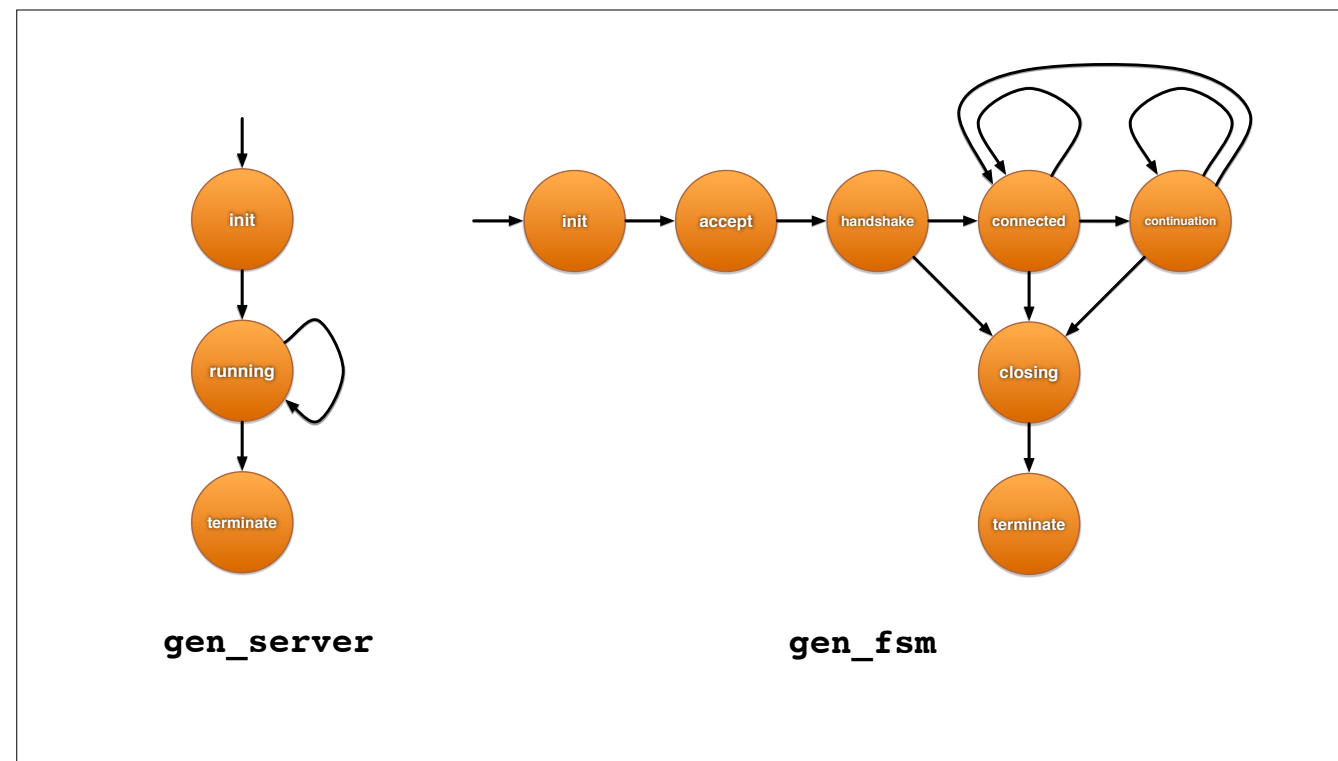
# The Generic Finite State Machine

code name: **gen\_fsm**

**AW YEAH!!!**

**gen\_server** didn't work out for me, and I learned that lesson pretty quickly. Since protocols more complicated, and they do different things based on things that happened in the past, I moved to **gen\_fsm**.

**gen\_fsm** adds states and transitions to **gen\_server**, while keeping Erlang's messaging abstracted. We could have done the same thing to **gen\_server**, but Erlang did it for us!



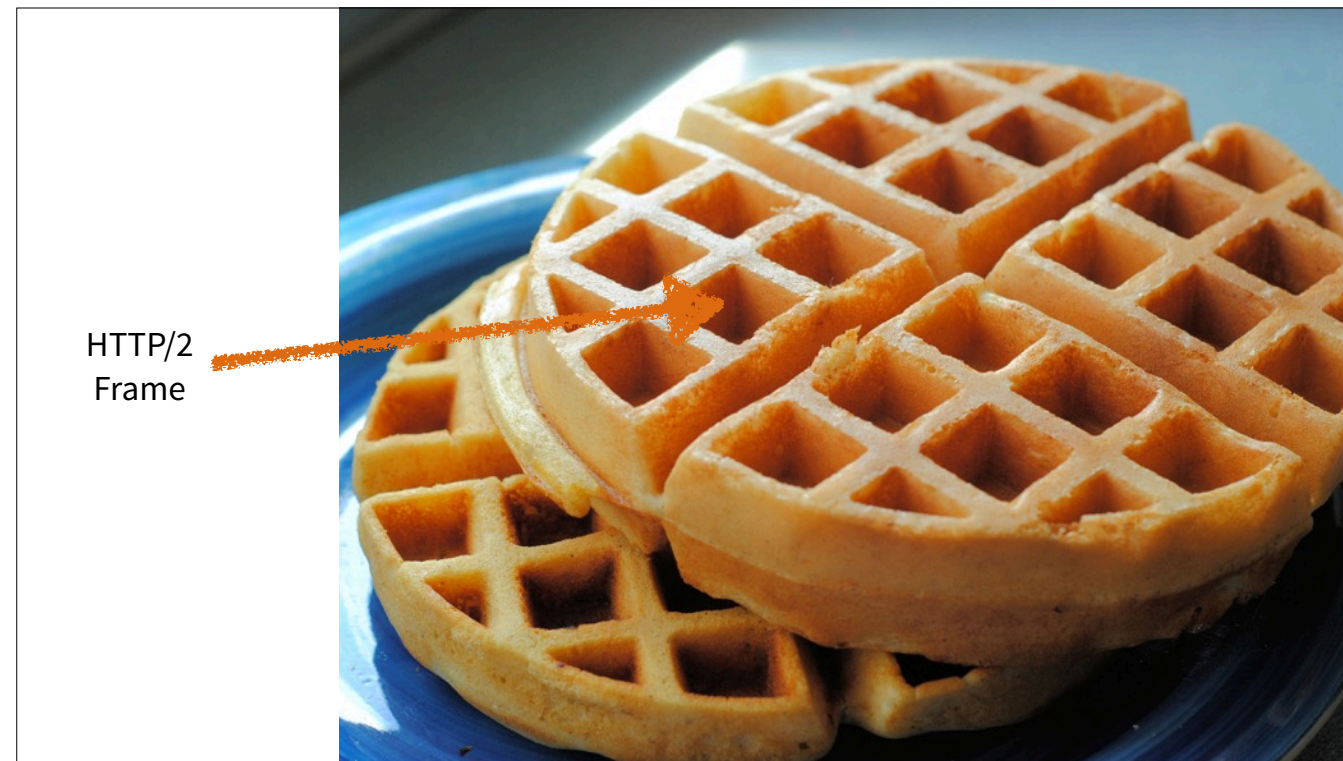
When you think about it, **gen\_server** is like a single state **gen\_fsm**, if you don't count **init** and **terminate**, which I don't because they can't receive process messages.

**gen\_fsm** gives you multiple "running", but acts different in each.

The word is "**gen\_fsm** is great for protocols". So step 1 is to start a bunch of them, but what is step 2?



Step 2? We need to know more about the protocol for that, so let's dive in!



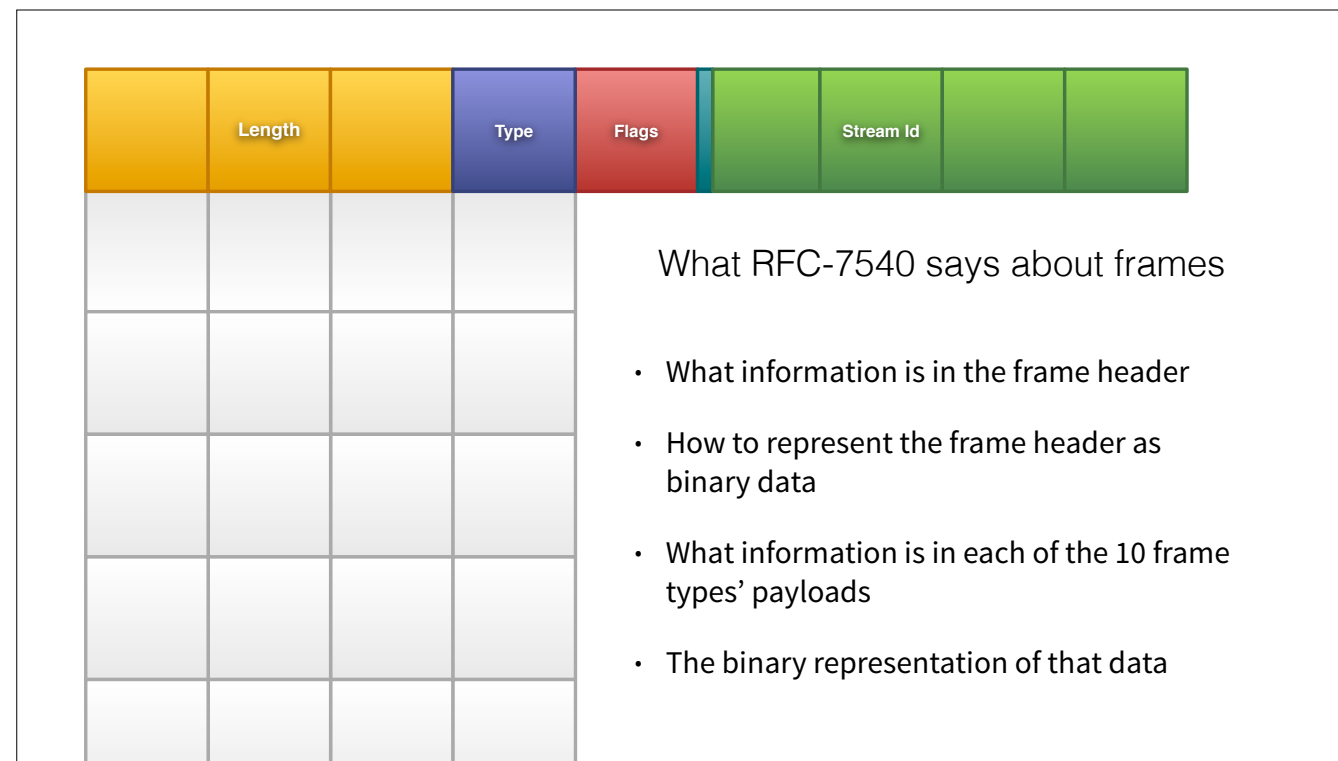
HTTP/2  
Frame

The HTTP/2 Frame is the most basic part of HTTP/2.

It's a data type!

It's a serialization format!

It's these little pockets on the waffle, which we can fill with all kinds of good different stuff!



But we don't want to fill the pockets with bad stuff. So the spec defines what we can put in the pockets.

It starts with a 9-byte header, but the frame type tells us about syrup and butter and ice cream and peanut butter. Making sure we don't fill the pockets with rocks or chalk or gross things to eat.

The three byte length in the header tells us how many bytes the grey payload is. The Flags basically tell us certain routing information.

The stream id is 31 bits, which means  $2^{31}-1$  possible streams per connection.

## Reading a Frame

```
{ok, FrameHeaderBin} = Transport:recv(Socket, 9),  
  
FrameHeader =  
    http2_frame:read_binary_frame_header(FrameHeaderBin),  
  
{ok, FramePayloadBin} =  
    Transport:recv(Socket, FrameHeader#frame_header.length),
```

How do we read frames in Erlang? This simple thing reads 9 bytes off the socket. That's it!

## Reading a Frame

```
{ok, FrameHeaderBin} = Transport:recv(Socket, 9),  
  
FrameHeader =  
    http2_frame:read_binary_frame_header(FrameHeaderBin),  
  
{ok, FramePayloadBin} =  
    Transport:recv(Socket, FrameHeader#frame_header.length),
```

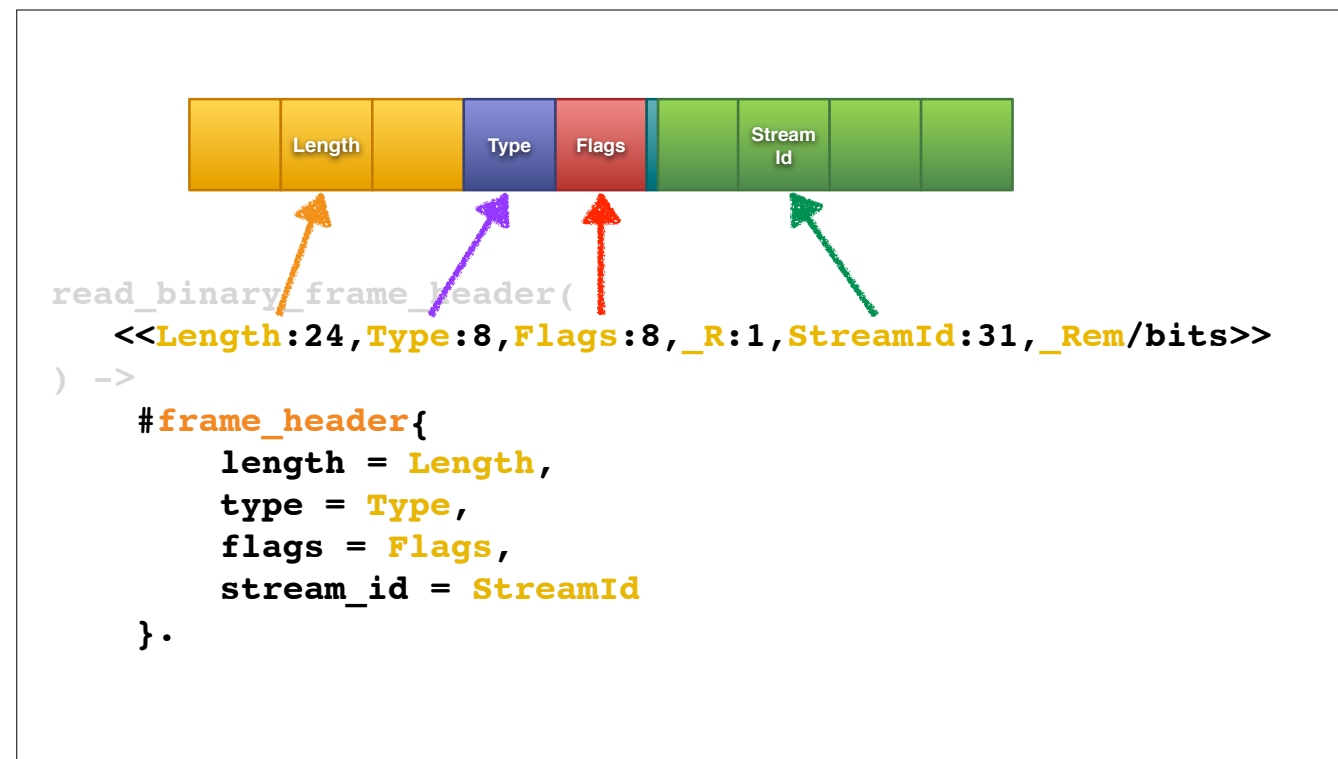
This deserializes them into a record we can use

## Reading a Frame

```
{ok, FrameHeaderBin} = Transport:recv(Socket, 9),  
  
FrameHeader =  
    http2_frame:read_binary_frame_header(FrameHeaderBin),  
  
{ok, FramePayloadBin} =  
    Transport:recv(Socket, FrameHeader#frame_header.length),
```

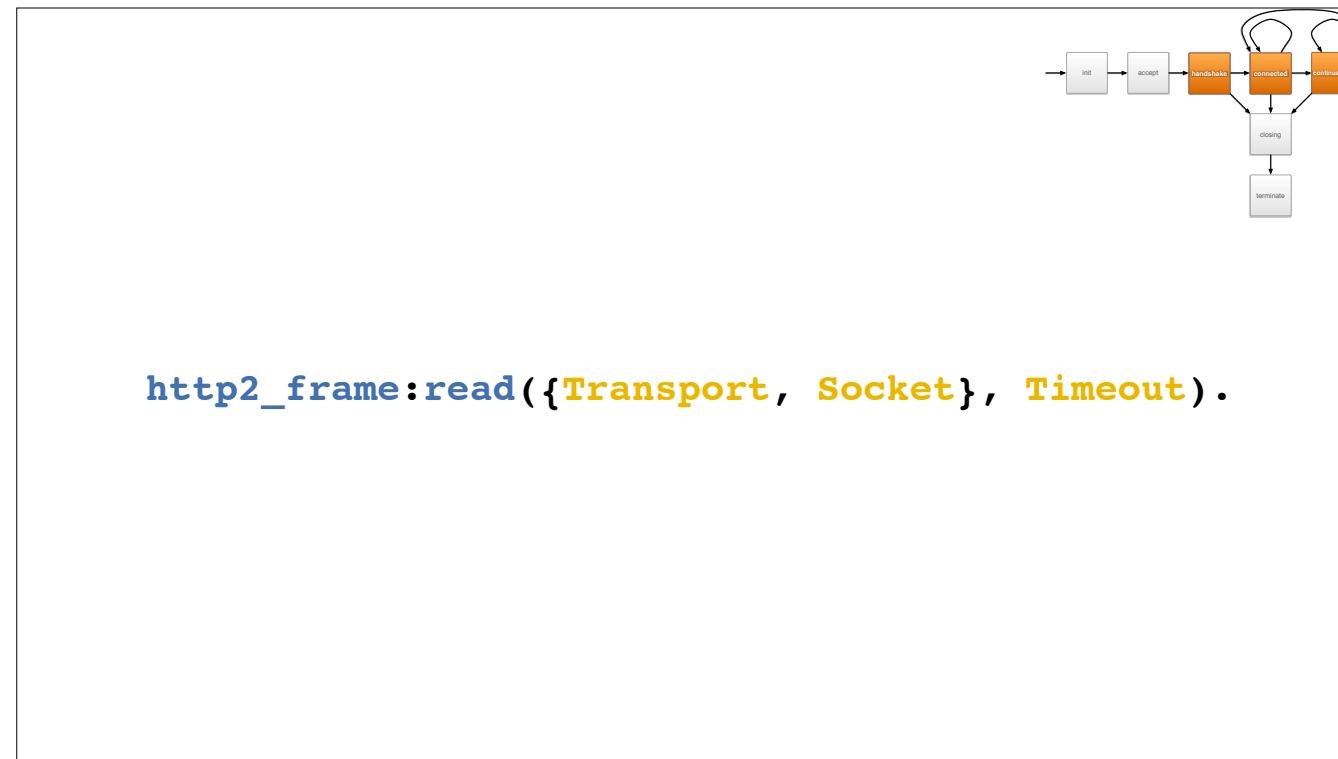
Then we use that record to pull the length out and read that many bytes  
[not shown] deserialize payload binary, too much code for one slide



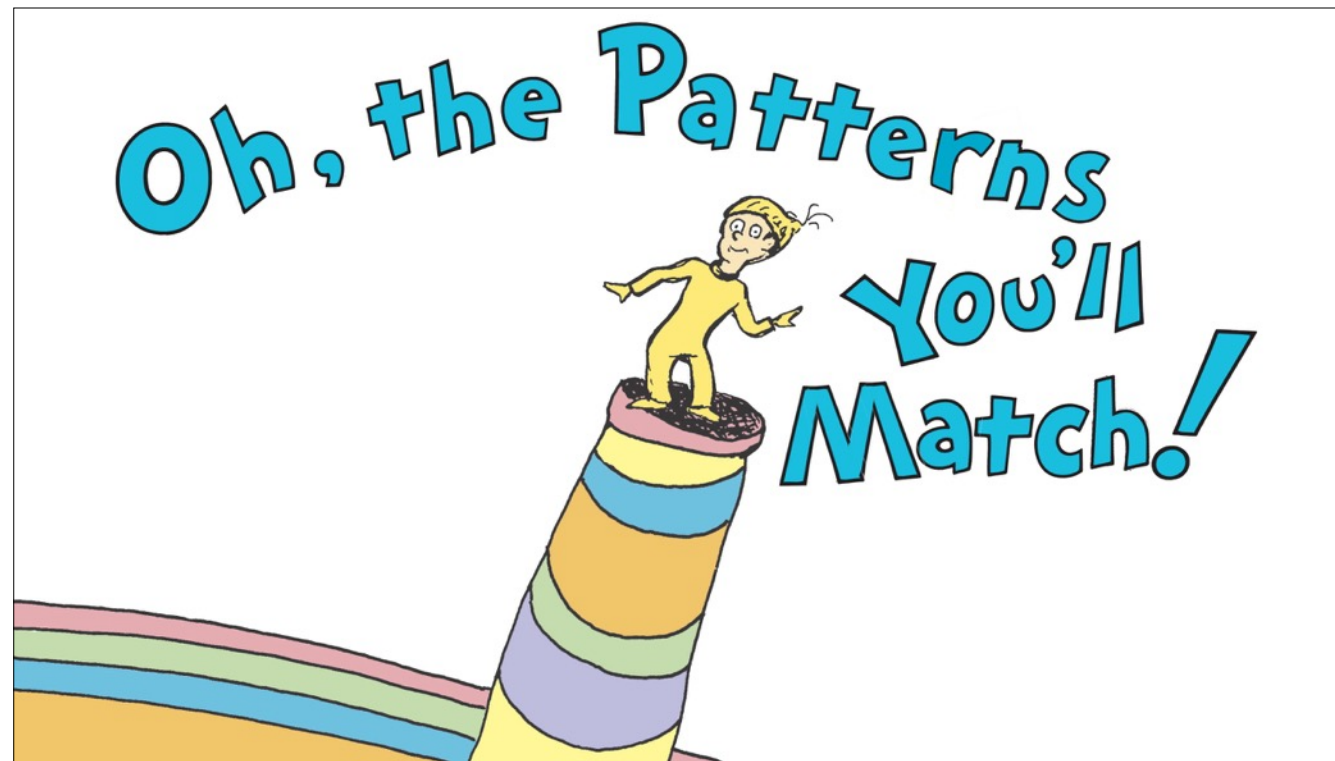


This is the `read_binary_frame_header` function. It takes in a binary as an argument and then we use binary pattern matching to bind these specific bits to big-endian unsigned integers. We know it's an Erlang binary because it's wrapped in chevrons, like `<<this>>`. The first 24 bits are bound to **Length**, the second 8 to **Type**, the next 8 to **Flags**. Then we throw away one bit, and the last 31 are bound to **StreamId**. We can use these values to construct a `#frame_header{}` record.

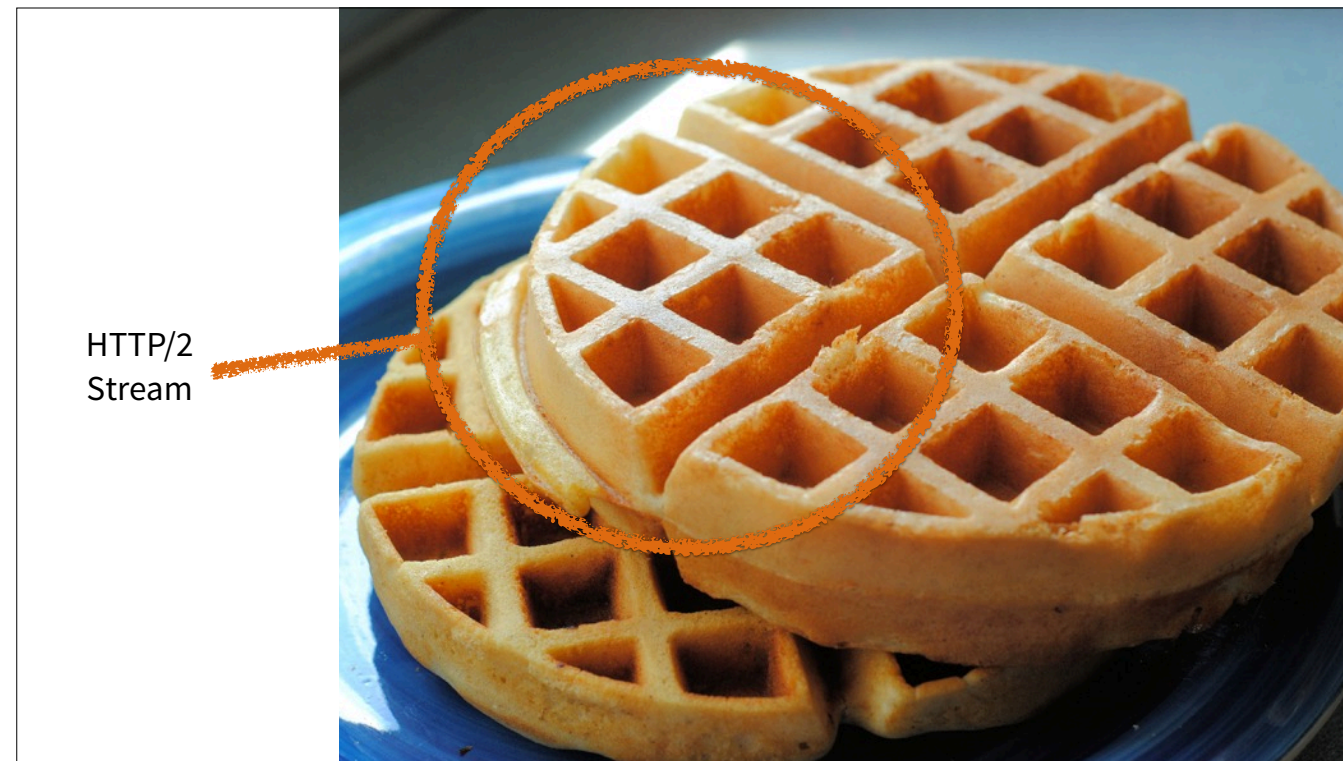
Yes! It's that easy.



Those are all internally used functions. We just wrap it up and use it like this in our FSM.

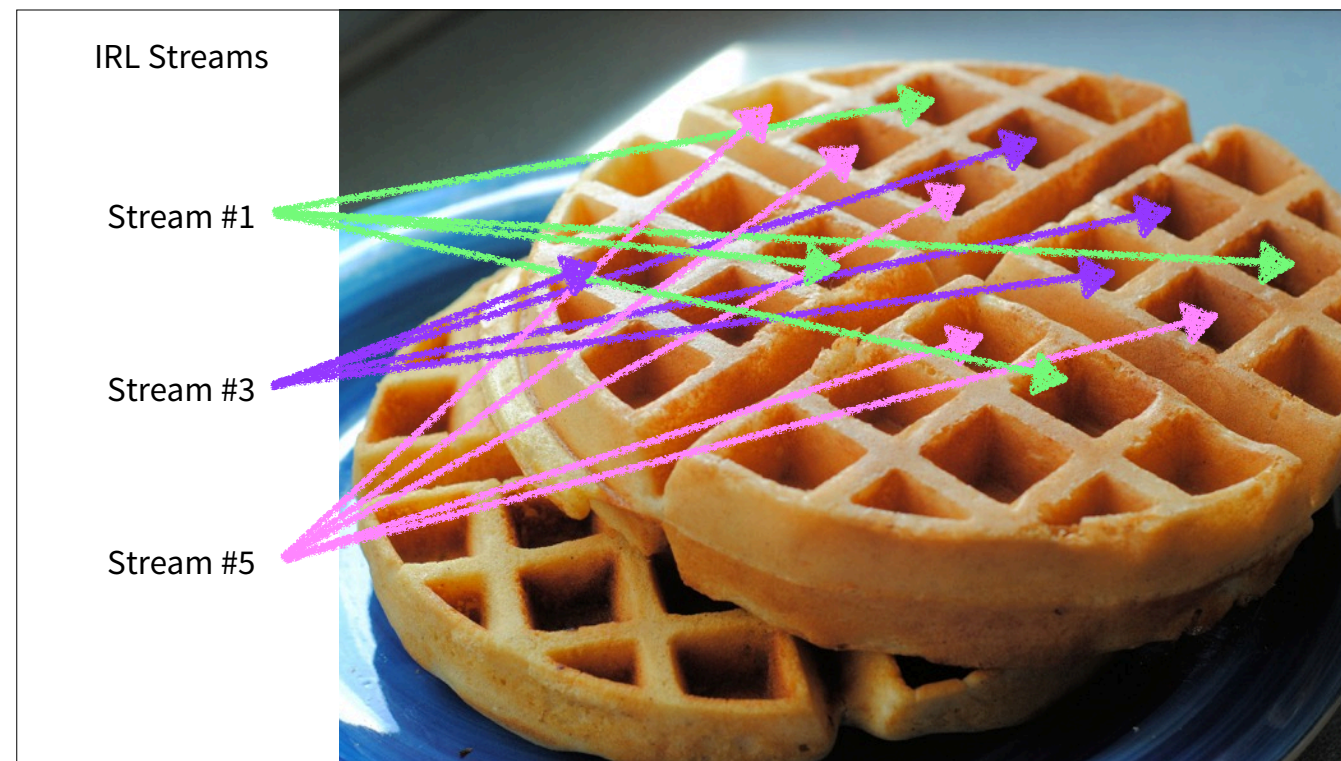


Isn't pattern matching great? It's one of those killer parts of Erlang's syntax.



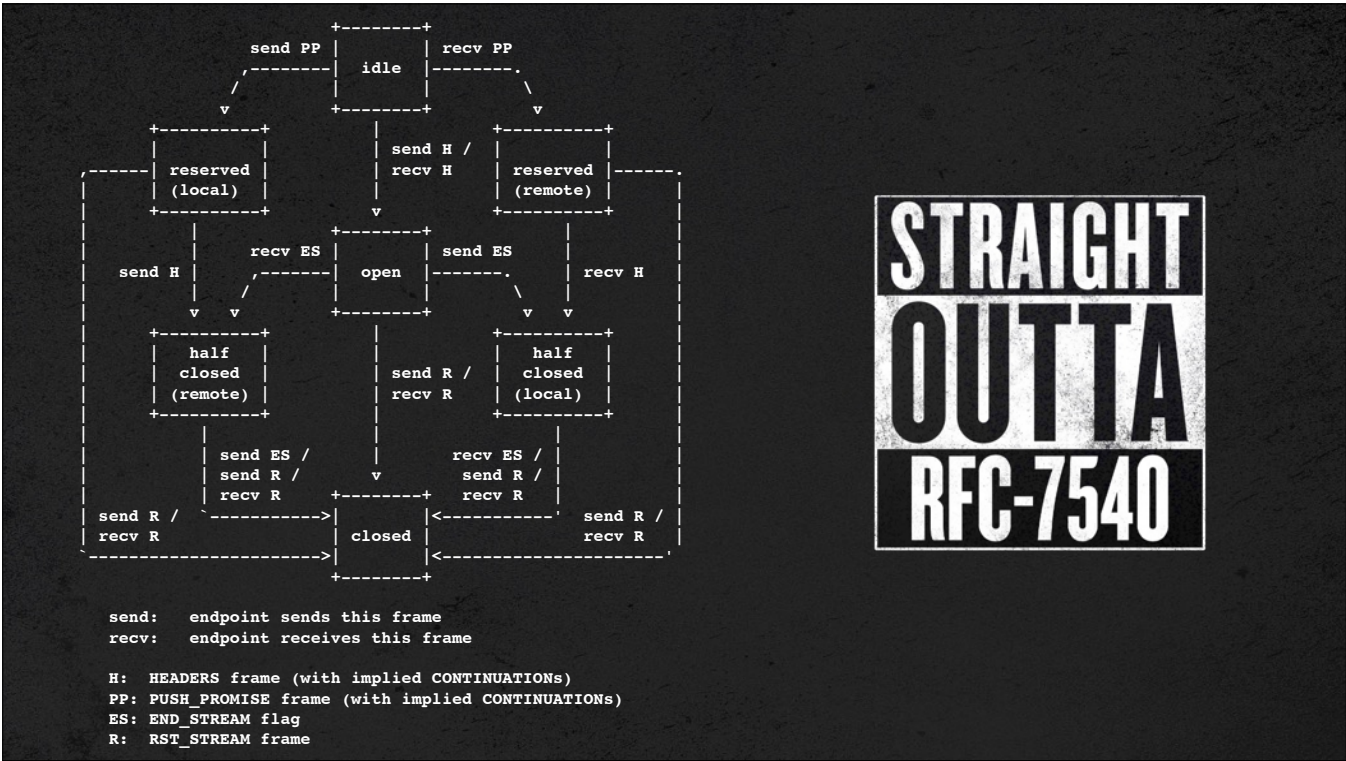
HTTP/2  
Stream

Taking a step up the data structure, we've got a stream. Physically, streams are sequences of frames with the same StreamId. Logically, they each represent a single HTTP request & response.



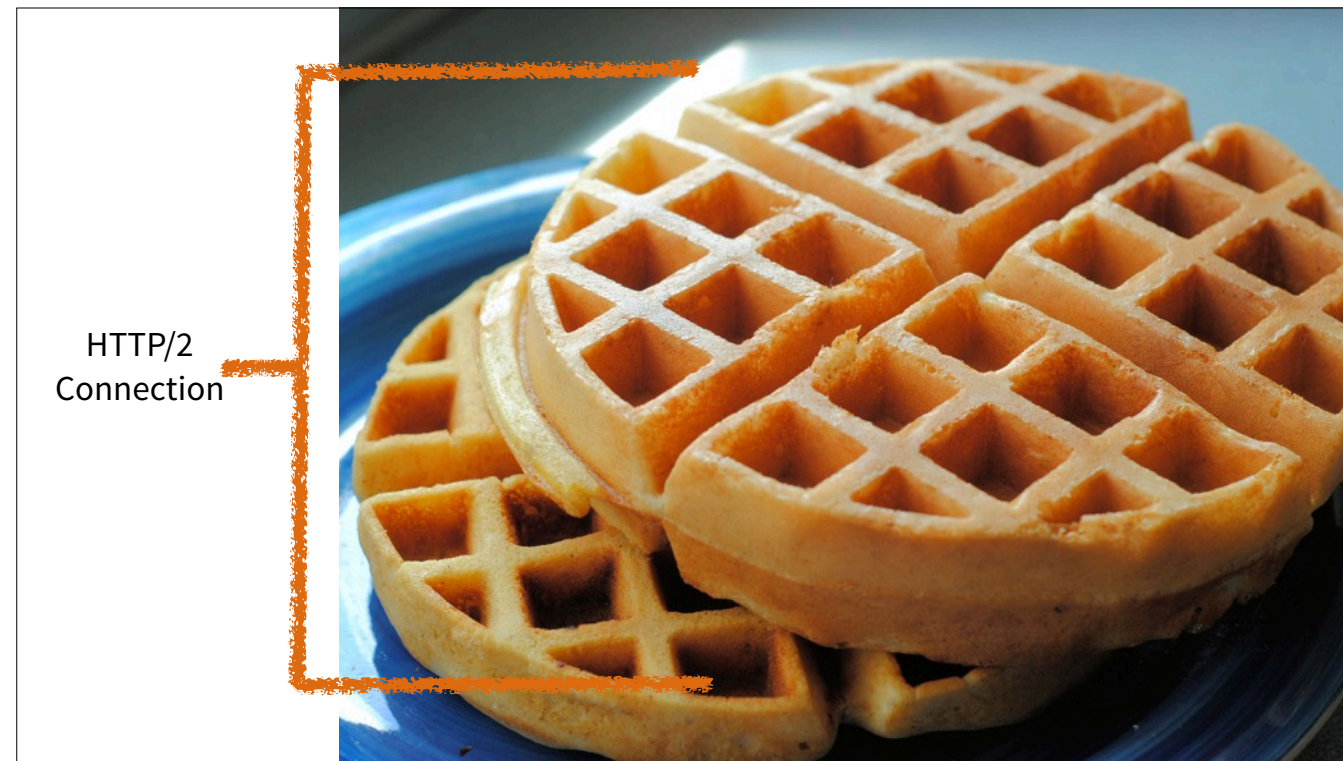
In real life they're not as simple as being a slice of the waffle. They come in over the single TCP connection interleaved, and it's up to our `gen_fsm` to reconstruct them by stream id.

Frame types and flags let it know when it's done, or when to take other actions.



A stream is also modeled as a finite state machine, but it's not one that we need the Erlang messaging semantics for. We'll take advantage of Erlang's referential transparency for applying single frames to a stream state record, transitioning through this fsm. The spec provides all kinds of rules for what types of frames are allowed to be processed in each of these states. For example, idle can only send/receive a HEADERS frame or a PUSH\_PROMISE frame.





The HTTP/2 Connection is like the plate. It's a persistent context for the meal... I mean connection. HTTP/2 remains a stateless protocol, but we're maintaining some state here for the sake of efficiency on the wire. Our `gen_fsm` maintains this state, allowing us to process individual requests in a stateless fashion.

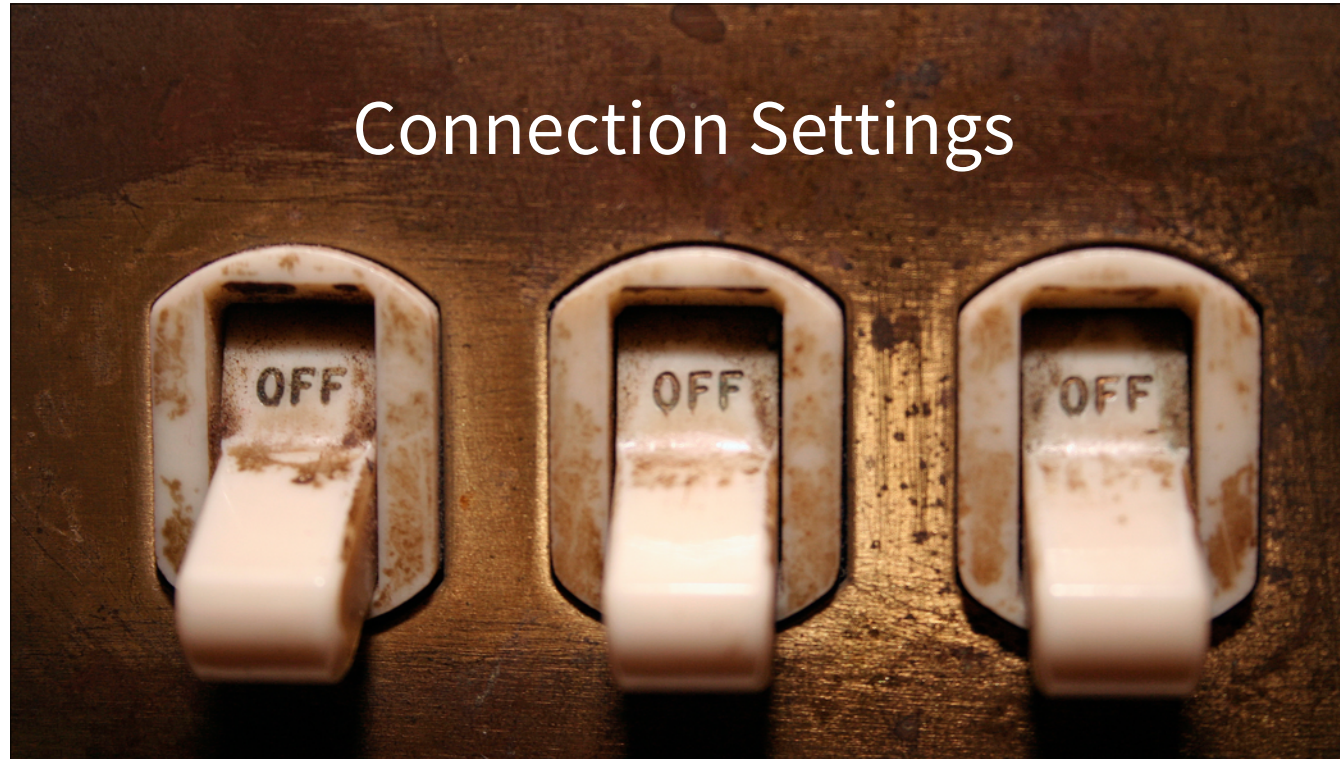
# The Short Version

- **frame:** data structure and serialization
- **stream:** semantic http request/response
- **connection:** persistent context for multiplexed streams

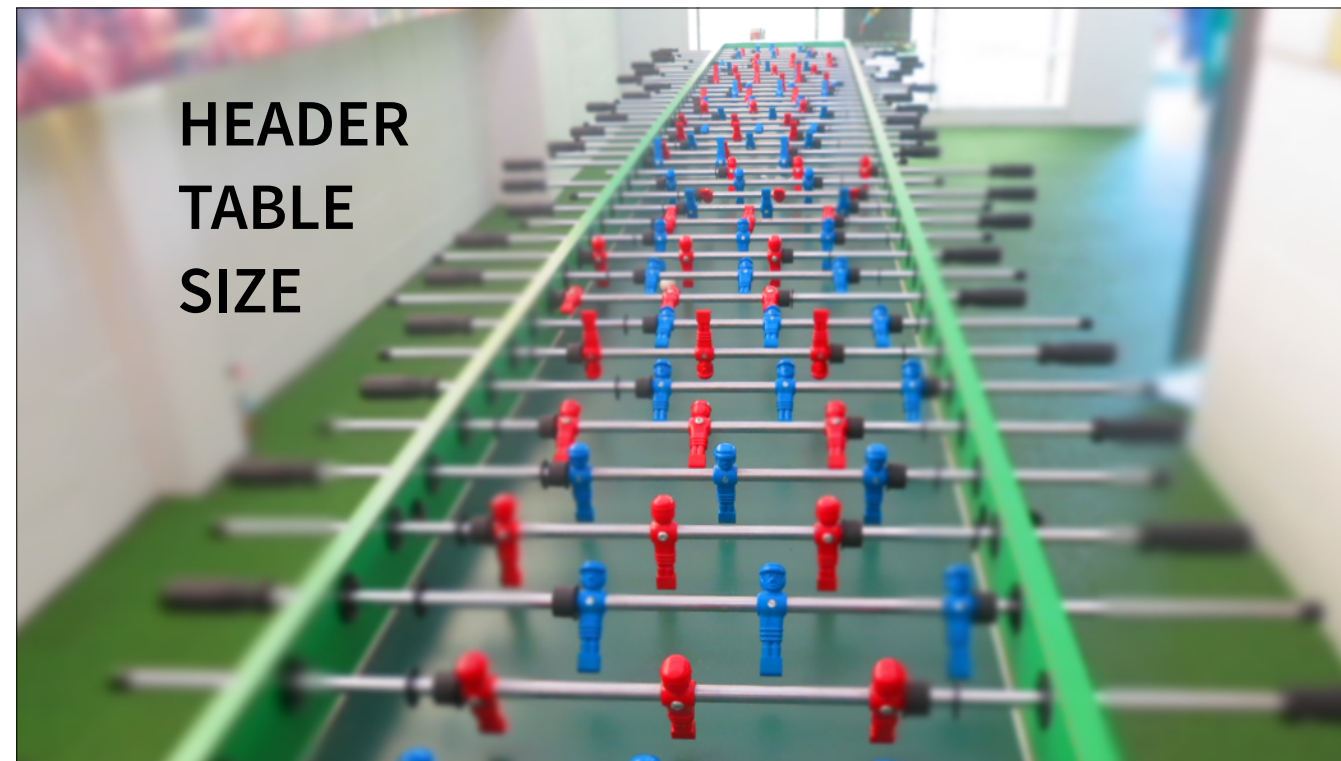
Let me explain. No, there is too much. Let me sum up. Buttercup is marry Humperdinck in little less than half an hour. So all we have to do is get in, break up the wedding, steal the princess, make our escape... after I kill Count Rugen.



## Connection Settings



The HTTP/2 has 6 settings that affect frame processing. The server & client both send values for these on connection as part of the settings handshake. This helps each side of the connection manage expectations. This is good, it's the foundation of any healthy relationship. These can be renegotiated at anytime as long as both sides agree

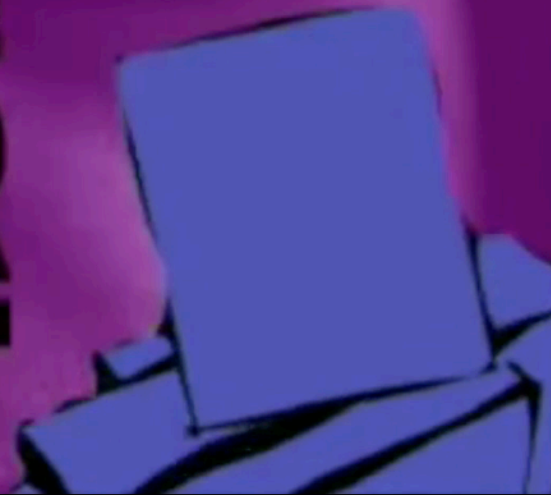


This is how big the header table can be. What's header table? This is all about HPACK, the HTTP/2's header compression protocol. Yo dawg, I heard you like protocols, so I put a protocol in your protocol.

To save space on wire, client and server keep indexes of headers, and when sending the same header again, will send some smaller value over the wire representing the index. These indexes have to live in state.

# ENABLE\_PUSH

**I AM THE  
PUSHER  
ROBOT**



Push Promises are super cool. Imagine you can send multiple responses to a single request! You ask for HTML, but I know that HTML will ask for this CSS and that JS and the other JPEG, so I send them all to you before you even know you want them. You're welcome!





Remember that ASCII stream FSM? All  $2^{31}-2$  streams begin in the idle state and end in the closed state. There are 5 states in between and this is how many streams can be in those 5 states at once.



HTTP/2 brings flow control to HTTP. It's a credit based scheme that puts a credit limit on how much data each side can send the other. They can be different for each direction! The receiver of data determines how much it will allow. This only applies to DATA frames, which represent HTTP request and response **bodies** only. As you send/receive bytes, you decrement the counter. and when it hits 0, you stop sending. If you receive more after that, you can throw an error. Initial window size is your initial credit balance, and when you run out, you can issue more credit with WINDOW\_UPDATE frames.

These credit balances are maintained at both the connection level and the stream level.

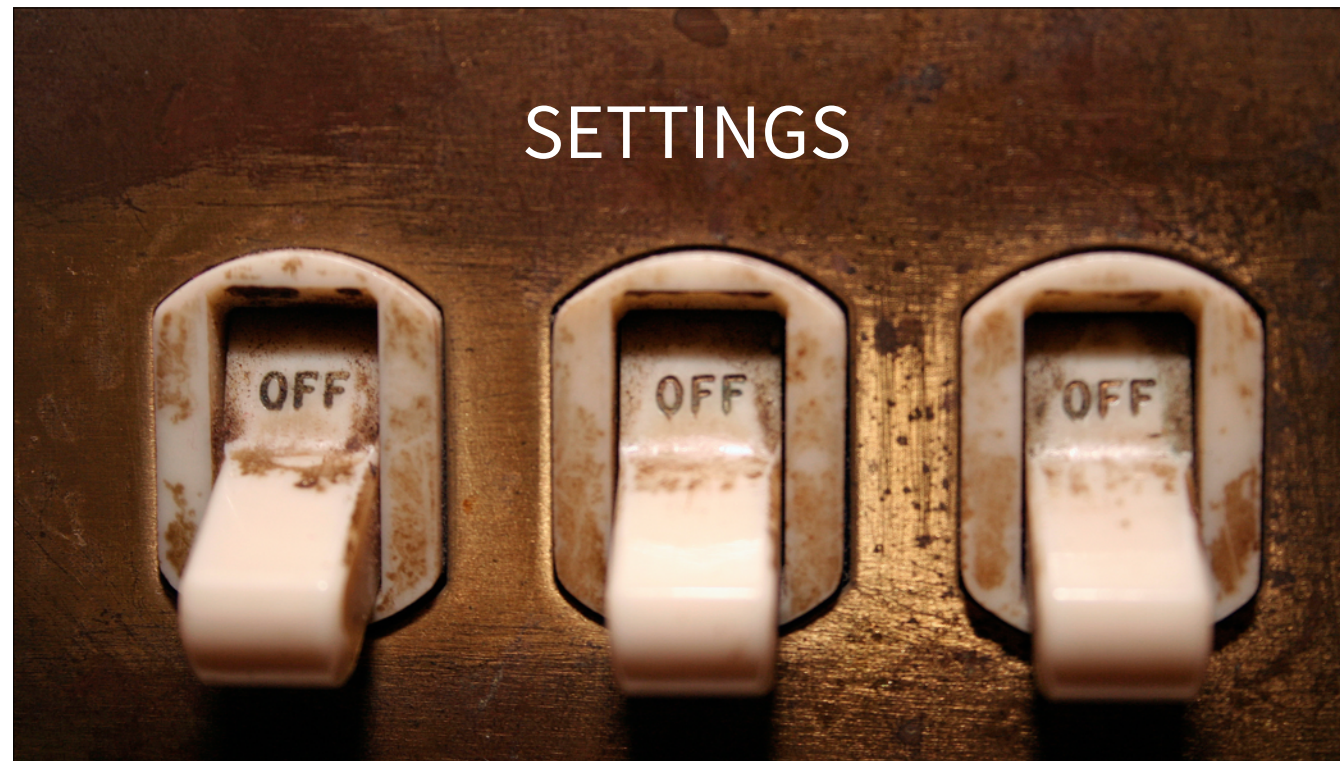




Frames have a length. We can set a maximum value on the length, and tune that along with TCP's MTU to work together on optimizing the network.



Max header list size is just a stream level cap for size of headers data.



Wow. That's almost the whole protocol. We talked about

- \* multiplexing
- \* flow control
- \* header compression
- \* server push

The only thing we left out was stream priority which is optional, so we'll ignore it for the sake of time.

These settings imply even more state





We'll have to have a way of tracking:

- how many streams are active
- what our current credit balance is at the connection level
- what our current credit balance is at the stream level as well as the state of each stream
- a table of request headers received (decode context)
- a table of response headers sent (encode context)

By the way, each peer can have different values for these settings, so we'll actually need to keep track of the server's to know what we can receive and the client's to know what we can send.

We can store these all in an Erlang state record, and that record is passed around as an argument to all **gen\_fsm**'s callbacks. What's a **gen\_fsm** callback?

# gen\_fsm callbacks

- |                              |                         |
|------------------------------|-------------------------|
| • <b>init/1</b>              | <b>StateName/2</b>      |
| • <b>terminate/3</b>         | • <b>handshake/2</b>    |
| • <b>handle_event/3</b>      | • <b>connected/2</b>    |
| • <b>handle_sync_event/4</b> | • <b>continuation/2</b> |
| • <b>handle_info/3</b>       | • <b>closing/2</b>      |
| • <b>code_change/4</b>       |                         |

I'm glad you asked! **gen\_fsm** does the heavy lifting, but we'll need to write code for the callbacks that make our **gen\_fsm** special.

The six callbacks on the left are required, without them, the **gen\_fsm** won't work at all.

**init/1** has to return the **gen\_fsm**'s initial state.

There's also callbacks that need to be defined for each **StateName**, so let's we'll need to define functions for these four states on the right.

**handle\_info/3**  
**handle\_event/3**  
**StateName/2**  
Transition Responses

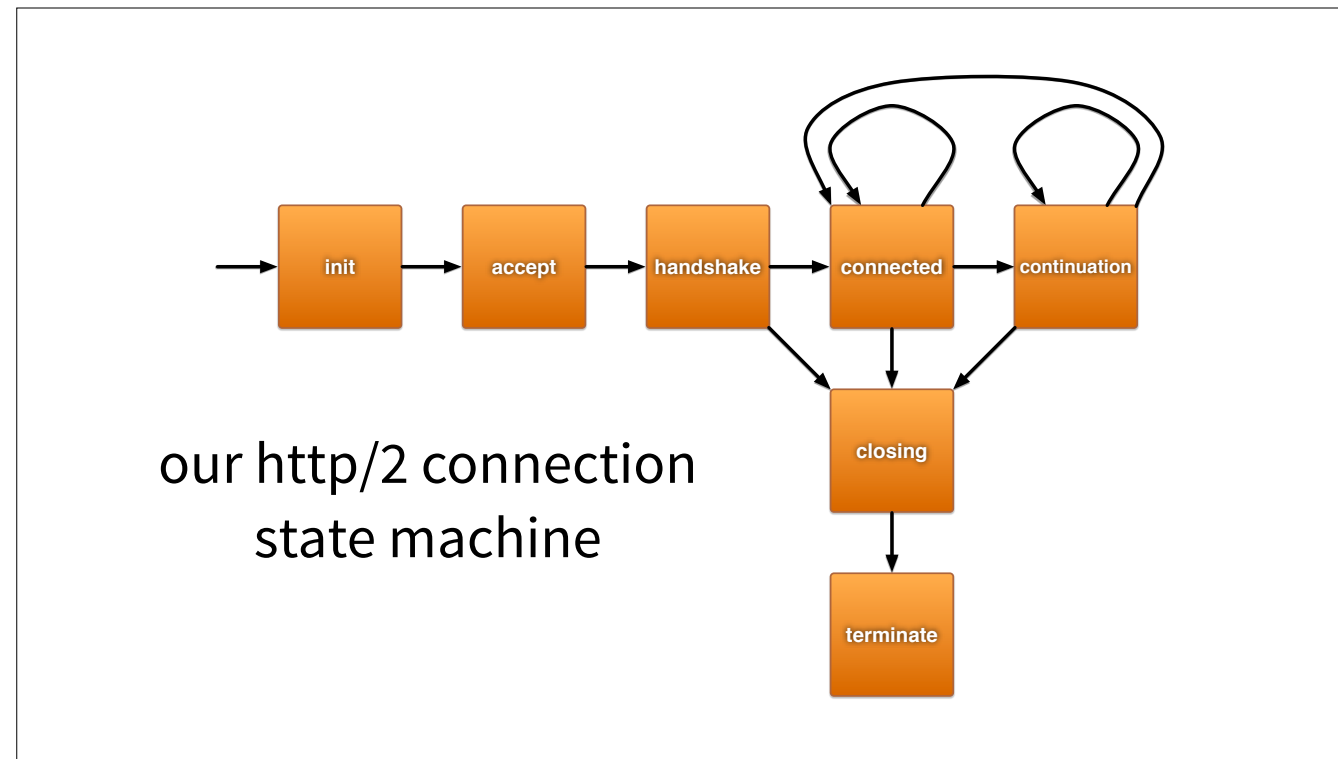
```
{next_state, NextStateName, NewStateData}
```

```
{next_state, NextStateName, NewStateData, Timeout}
```

```
{next_state, NextStateName, NewStateData, hibernate}
```

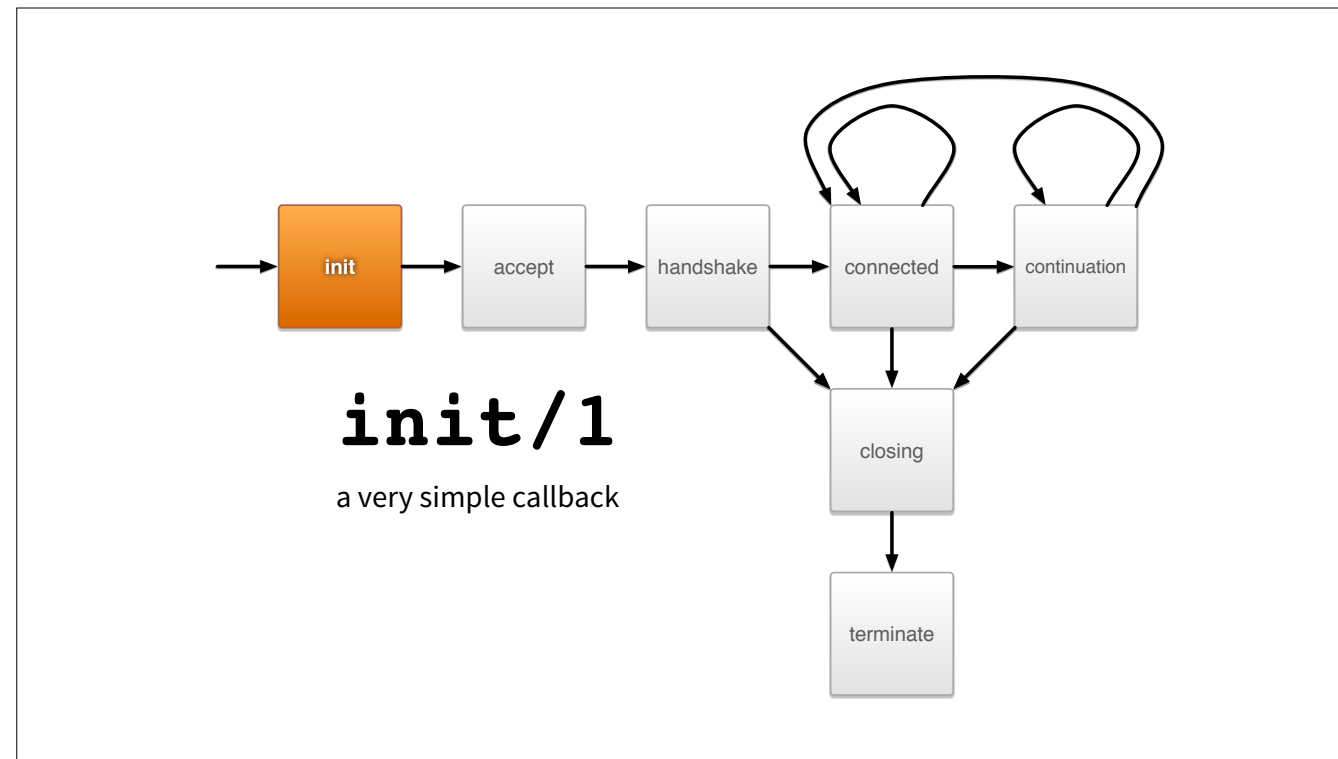
```
{stop, Reason, NewStateData}
```

The callbacks we're using have four possible returns, but there are even more for some other callbacks. These are the only things these functions can return. We're most interested in the second and last. We'll use the first one in the handshake callback



Aside from the init and terminate states, we've got:

- \* accept - where we hang waiting for clients
- \* handshake - where we establish that it's an HTTP/2 connection
- \* connected - where we process frames
- \* continuation - one weird hpack race
- \* closing - just close a socket

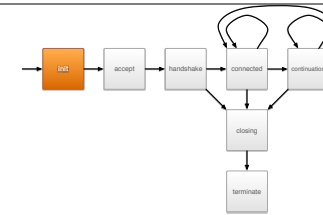


**init/1** runs one time on start up, it's the callback that is triggered when the supervisor creates a new process. It takes one arg, usually list.

```

init([Transport, ListenSocket],
     SSLOptions]) ->
{ok, Ref} =
  prim_inet:async_accept(ListenSocket, -1),
{ok,
 accept,
 #connection_state{
   listen_ref=Ref,
   socket = {Transport, undefined},
   ssl_options = SSLOptions
 }}.

```

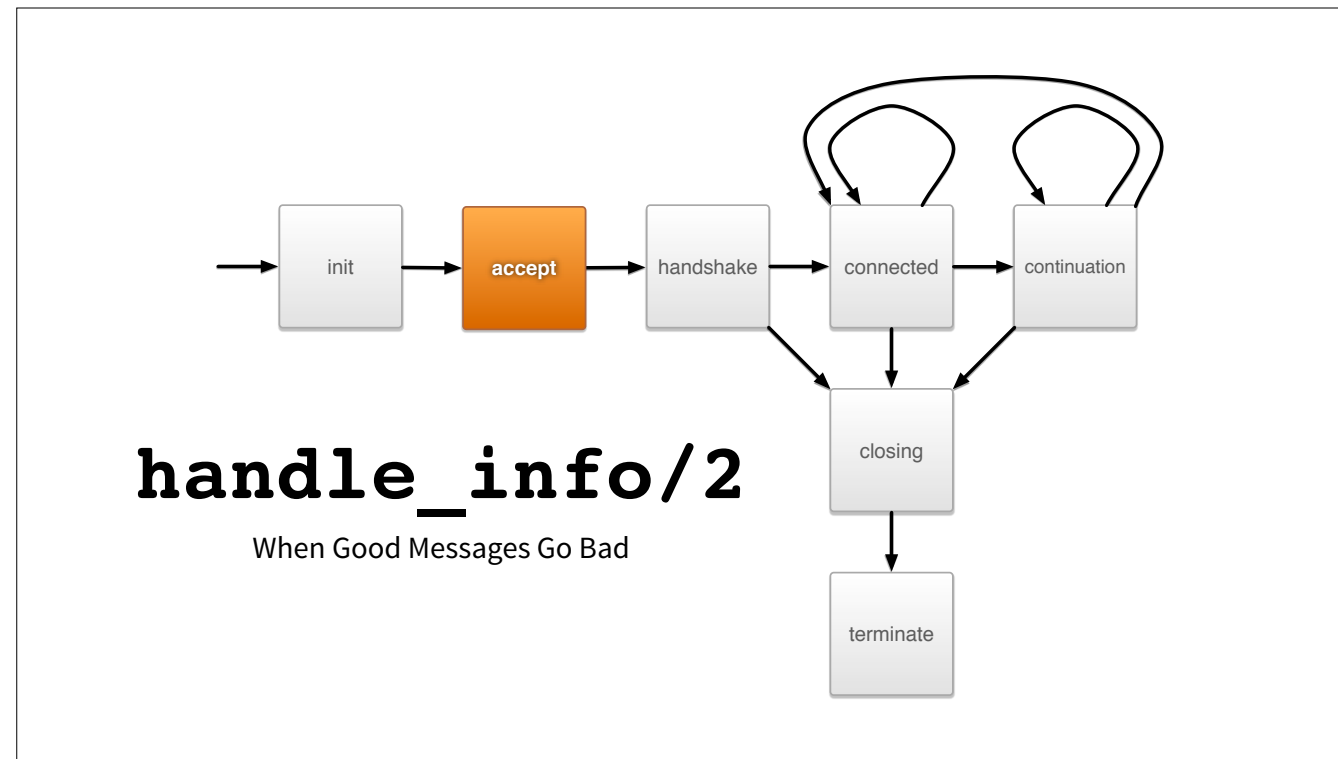


[Director's Cut] This slide wasn't in the talk, but it's the simple implementation of the `init/1` callback

What we're saying here is to go ahead and just spawn an acceptor somewhere. When a connection comes along we'll get a process message that let's us know and we'll deal with it then. We'll store some of this information in the FSM's state, and then transition into the accept state, where we wait for a client to come along.



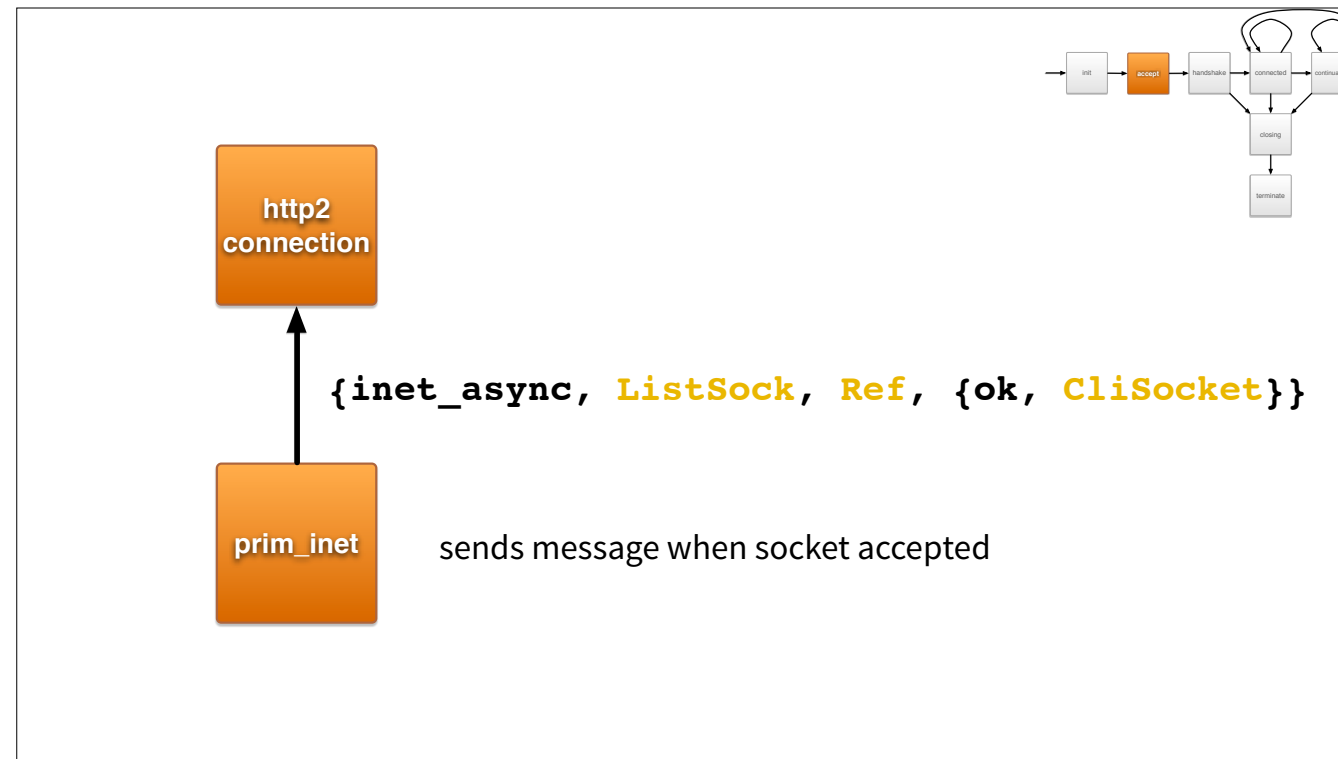
This `prim_inet` call tells Erlang to let us know when a client connects, and we'll do our own thing until then. More info at <http://joedevivo.com/2015/09/06/prim-inet.html>



**handle\_info/2** is for handling messages that are not managed by the **gen\_fsm** messaging API.

When that socket lets us know a client has connected, it does so in a non-gen\_fsm way





any process that calls `prim_inet:async_accept/2` will get the same type of message back!

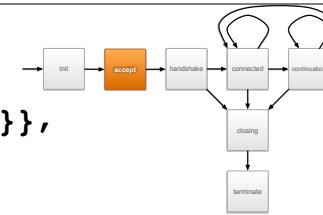
This is good! we can use this anywhere, not exclusive to `gen_fsm`

```

handle_info(
  {inet_async, _ListSock, Ref, {ok, CliSocket}},
  accept,
  S=#connection_state{
    ssl_options = SSLOptions,
    socket = {Transport, undefined},
    listen_ref = Ref
  }) ->
%%Socket = Accepted Socket after various socket functions
chatterbox_sup:start_socket(),

{next_state,
 handshake,
 S#connection_state{
   socket = {Transport, Socket}
 },
 0};

```



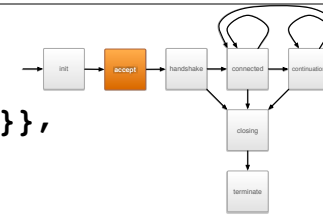
Our handle info function is a good example of function clause matching and record syntax, so let's talk about them

```

handle_info(
  {inet_async, _ListSock, Ref, {ok, CliSocket}},
  accept,
  S=#connection_state{
    ssl_options = SSLOptions,
    socket = {Transport, undefined},
    listen_ref = Ref
  }) ->
%%Socket = Accepted Socket after various socket functions
chatterbox_sup:start_socket(),

{next_state,
 handshake,
 S#connection_state{
   socket = {Transport, Socket}
 },
 0};

```



This line is the pattern of the message we saw **prim\_inet** sending. If it doesn't match, we don't run this clause. Matching means "It's a 4-tuple that starts with **inet\_async**, can have anything in elements 2 and 3, and element 4 has to be a 2-tuple that starts with **ok**"

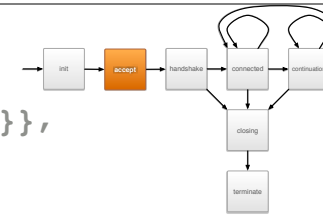
If it does, we bind **Ref** and **CliSocket**. We throw away the value in the second element. That's what the underscore is all about.

```

handle_info(
  {inet_async, _ListSock, Ref, {ok, CliSocket}},
  accept,
  S=#connection_state{
    ssl_options = SSLOptions,
    socket = {Transport, undefined},
    listen_ref = Ref
  }) ->
%%Socket = Accepted Socket after various socket functions
chatterbox_sup:start_socket(),

{next_state,
 handshake,
 S#connection_state{
   socket = {Transport, Socket}
 },
 0};

```



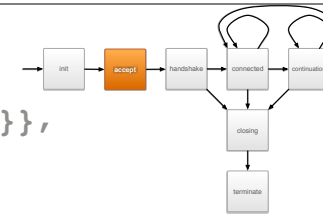
This line in the function clause pattern means only run this clause if we're in the **accept** state

```

handle_info(
  {inet_async, _ListSock, Ref, {ok, CliSocket}},
  accept,
  S=#connection_state{
    ssl_options = SSLOptions,
    socket = {Transport, undefined},
    listen_ref = Ref
  }) ->
  %%Socket = Accepted Socket after various socket functions
  chatterbox_sup:start_socket(),

  {next_state,
   handshake,
   S#connection_state{
     socket = {Transport, Socket}
   },
   0};

```



Here we bind our current connection state record to **S**, but we also reconstruct the connection state record and bind **SSLOptions**, **Transport** and **Ref**. If it can't deconstruct the record because this argument isn't a record, then we don't match this function clause.

See how the **Ref** we bound on the first line is also yellow? That's because we used the same name for two bindings. Since this is Erlang, those two things need to equal. If they're not equal, this function clause doesn't match!

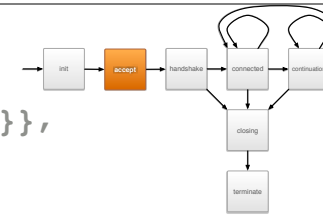
Also, the clause doesn't match if the second element of the connection state's socket field is not **undefined** because we don't want to accept a socket if we already did

```

handle_info(
  {inet_async, _ListSock, Ref, {ok, CliSocket}},
  accept,
  S=#connection_state{
    ssl_options = SSLOptions,
    socket = {Transport, undefined},
    listen_ref = Ref
  }) ->
%%Socket = Accepted Socket after various socket functions
chatterbox_sup:start_socket(),

{next_state,
 handshake,
 S#connection_state{
   socket = {Transport, Socket}
 },
 0};

```



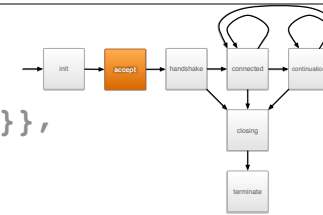
This line is telling the supervisor to factory up another acceptor process, this one is now spoken for since we've graduated to real connection

```

handle_info(
  {inet_async, _ListSock, Ref, {ok, CliSocket}},
  accept,
  S=#connection_state{
    ssl_options = SSLOptions,
    socket = {Transport, undefined},
    listen_ref = Ref
  }) ->
%%Socket = Accepted Socket after various socket functions
chatterbox_sup:start_socket(),

{next_state,
 handshake,
 S#connection_state{
   socket = {Transport, Socket}
 },
 0};

```



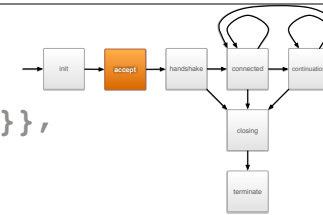
This is the tuple everything has to return in **gen\_fsm**, it's telling us to move to the next state, which is **handshake**, with this context, and send a timeout msg in 0ms. Why? We'll get to that.

```

handle_info(
  {inet_async, _ListSock, Ref, {ok, CliSocket}},
  accept,
  S=#connection_state{
    ssl_options = SSLOptions,
    socket = {Transport, undefined},
    listen_ref = Ref
  }) ->
  %%Socket = Accepted Socket after various socket functions
  chatterbox_sup:start_socket(),

  {next_state,
   handshake,
   S#connection_state{
     socket = {Transport, Socket}
   },
   0};

```



This is creating the new State record, which is like a copy of the state with the value for This syntax for records is saying take S, make a copy with the socket field changed to the new socket we accepted and registered with the Erlang VM.

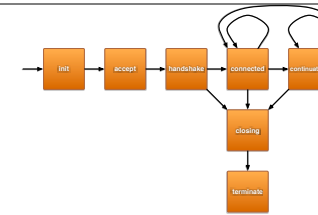
Why would we want to create a copy of the connection state record?





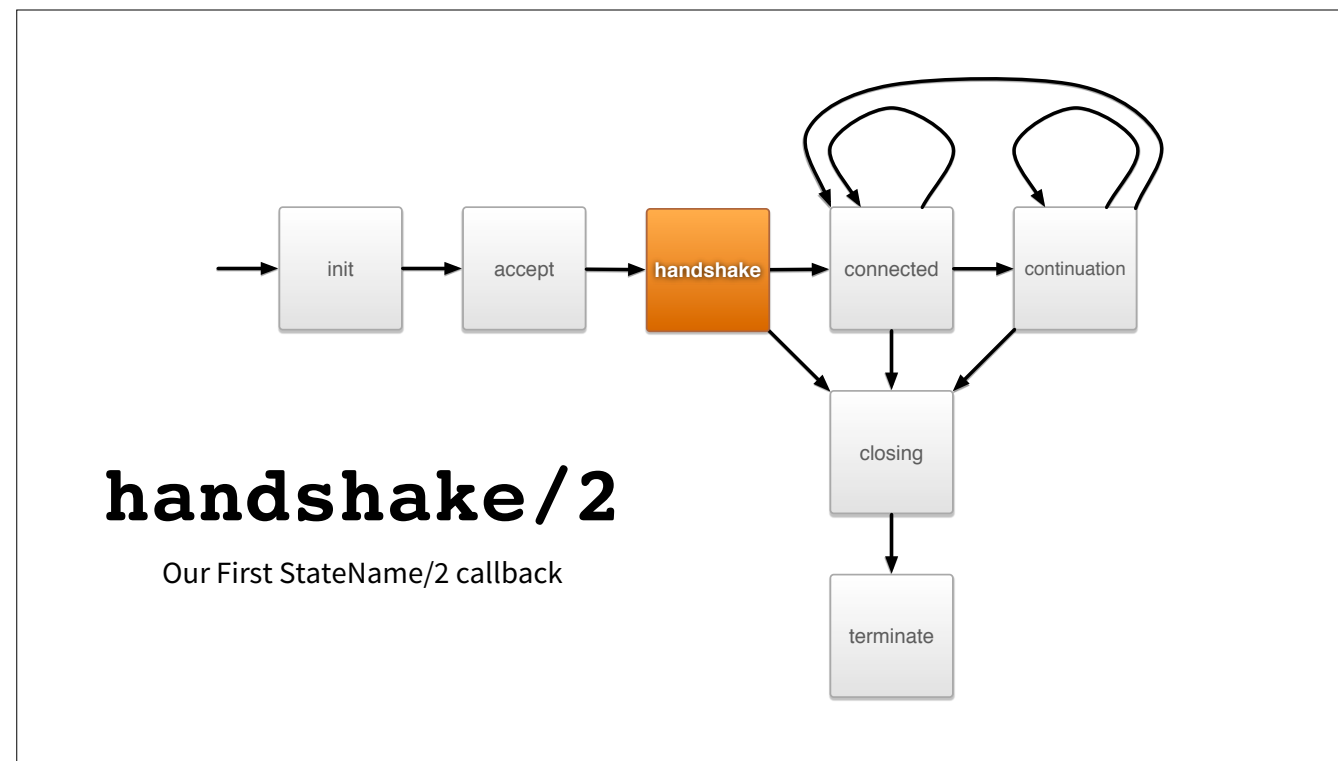
**BUT IT'S COOL  
'CAUSE WE'RE  
PART OF A TEAM  
[OF PROCESSES]**

Immutability takes some complexity out of concurrency, and since Erlang is all about concurrency, it leans into the immutability too.



```
handle_info({tcp_closed, _Socket}, _StateName, State) ->
{stop, normal, S};
```

[Director's Cut] This slide wasn't in the talk, but it's another clause of the **handle\_info/2** callback that receives a message if the socket closes. If it closes unexpectedly, it will send a message to our **gen\_fsm** (isn't Erlang cool?)

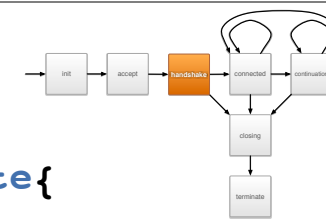


We didn't need a **StateName/2** callback for **accept** because it was handled by **handle\_info/2**

```

handshake(timeout,
           StateWithSocket=#connection_state{
             socket={Transport, Socket}
           }) ->
{ok, <<"PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n">>
 = Transport:recv(AcceptSocket, 24, 5000),
 {next_state, connected, StateWithSocket, 0}.

```



And now we have an FSM just sitting around with an open socket. Just opening a socket's never been so easy! Now what?

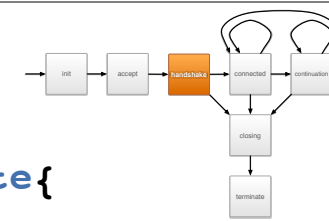
We read the first 24 bytes off the socket and make sure they're equal to the binary string **"PRI \* HTTP/2.0\r\n\r\nSM\r\n\r\n"**

What if it doesn't match? Well, this thing will explode with error messages and crash logs

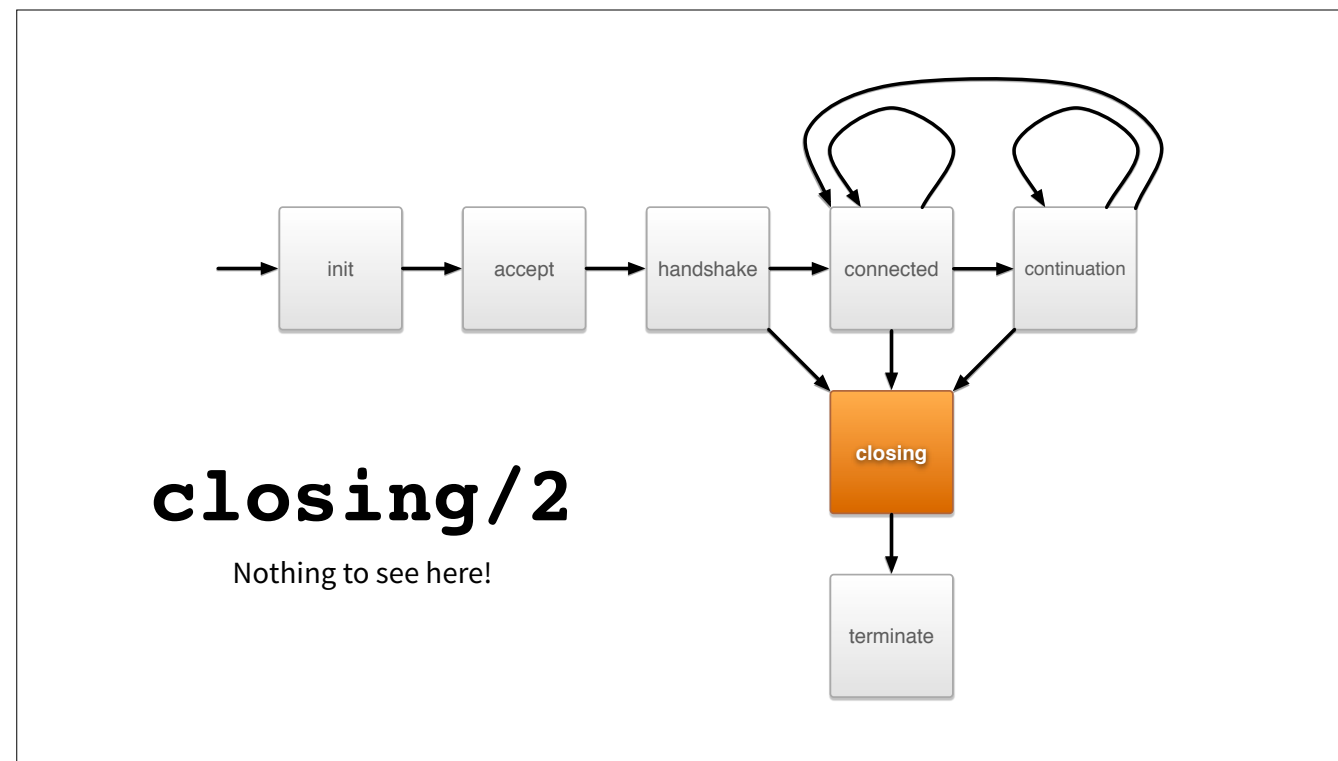
```

handshake(timeout,
    StateWithSocket=#connection_state{
        socket={Transport, Socket}
    }) ->
case Transport:recv(AcceptSocket, 24, 5000) of
    {ok, <<"PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n">>} ->
        {next_state, connected, StateWithSocket, 0};
    _ ->
        {next_state, closing, StateWithSocket, 0}
end.

```



We don't have to be jerks about it. We can put the pattern match in a **case** statement and exit cleanly via the **closing** state if it doesn't match. Let's talk about that one next.

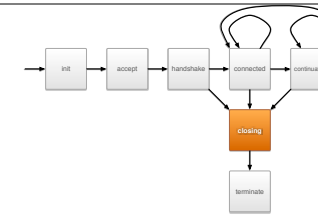


This is a place to clean up anything we want before letting Erlang have it's turn.

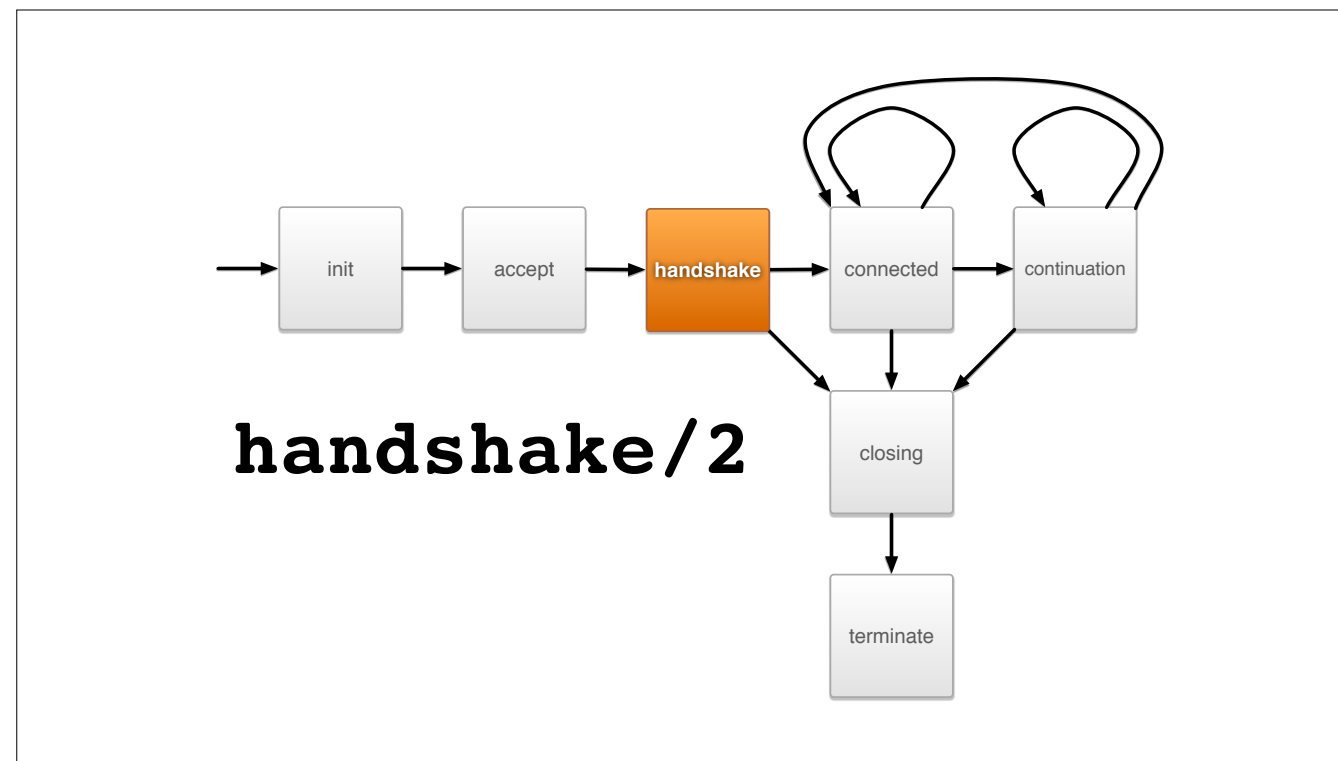
```

closing(Message, State=#connection_state{
    socket={_, undefined}
}) ->
    %% Does nothing if socket is undefined
    {stop, normal, State};
closing(Message, State=#connection_state{
    socket={Transport, Socket}
}) ->
    %% Closes the socket otherwise
    Transport:close(Socket),
    {stop, normal, State}.

```



We can use pattern matching in the function clause to decide whether or not we try and close the socket before stopping the server. Basically, if we don't have a socket defined, don't bother closing it.



Let's go back and finish that handshake properly



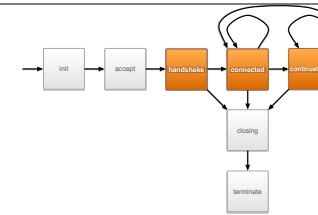


There's an HTTP/2 Frame called "go away" that tells the connection that we can't recover.  
There are lots of reasons this could happen, and some of them even have specific codes!

```

go_away(ErrorCode,
        State = #connection_state{
            socket={Transport,Socket},
            next_available_stream_id=NAS
        }) ->
GoAway = #goaway{
    last_stream_id=NAS,
    error_code=ErrorCode
},
GoAwayBin = http2_frame:to_binary({#frame_header{
    stream_id=0
}, GoAway}),
Transport:send(Socket, GoAwayBin),
{next_state, closing, State, 0}.

```

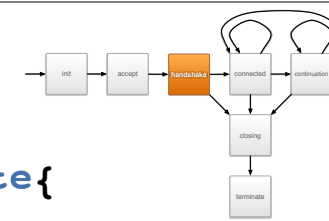


This function gets everything it needs to create an HTTP/2 GOAWAY frame, serialize it to binary, send that binary over the socket to the client, and then return the thing that **gen\_fsm** expects to transition to the closing **state**.

```

handshake(timeout,
    StateWithSocket=#connection_state{
        socket={Transport, Socket}
    }) ->
case Transport:recv(AcceptSocket, 24, 5000) of
{ok, <<"PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n">>} ->
    {next_state, connected, StateWithSocket, 0};
_ ->
    {next_state, closing, StateWithSocket, 0}
end.

```

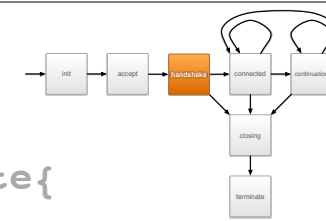


Here's the **handshake/2** callback, just as we left it.

```

handshake(timeout,
           StateWithSocket=#connection_state{
             socket={Transport, Socket}
           }) ->
case Transport:recv(AcceptSocket, 24, 5000) of
  {ok, <<"PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n">>} ->
    {next_state, connected, StateWithSocket, 0};
  _ ->
    {next_state, closing, StateWithSocket, 0}
end.

```

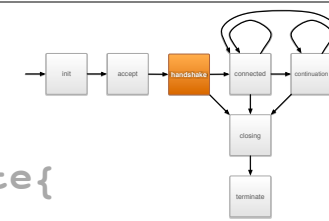


This is the line that we were using to transition to the **closing** state

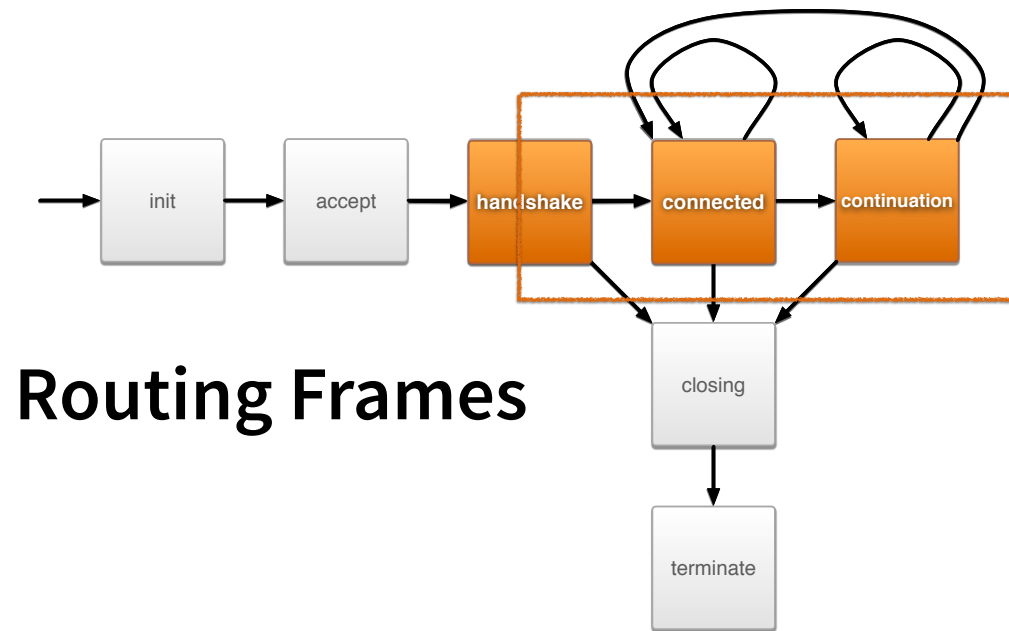
```

handshake(timeout,
           StateWithSocket=#connection_state{
             socket={Transport, Socket}
           }) ->
case Transport:recv(AcceptSocket, 24, 5000) of
  {ok, <<"PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n">>} ->
    {next_state, connected, StateWithSocket, 0};
  _ ->
    go_away(?PROTOCOL_ERROR, StateWithSocket)
end.

```



by using the **go\_away/2** function, we're sending a frame that HTTP/2 will understand AND returning a tuple that **gen\_fsm** will understand



## Routing Frames

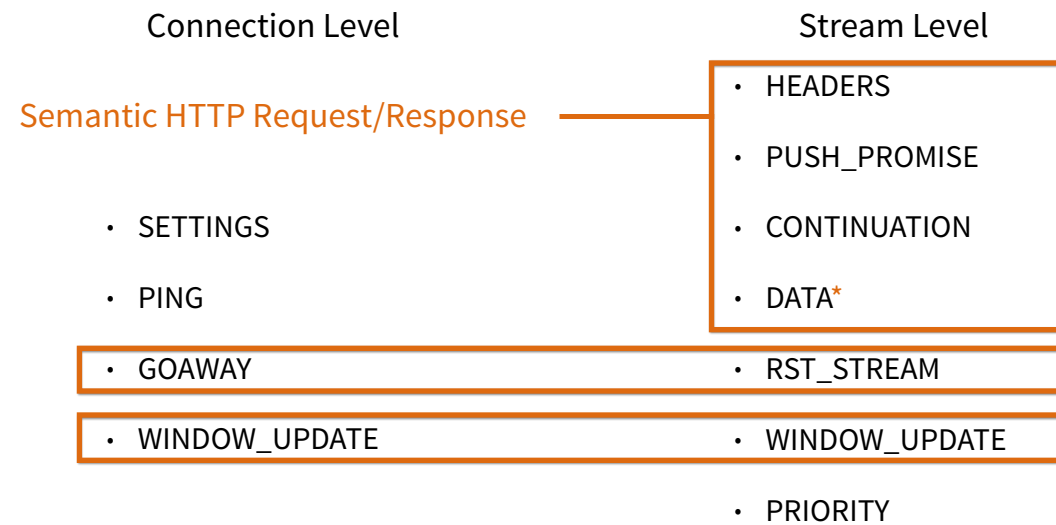
Once we've gotten the preamble, we're ready to route frames to their appropriate level. Everything that comes over the wire after that **"PRI \* HTTP/2.0\r\n\r\nSM\r\n\r\n"** will be a legitimate HTTP/2 frame.



# Routing Incoming Frames

We already know how to read one off the wire, so what do we do with them?

# Frame Types



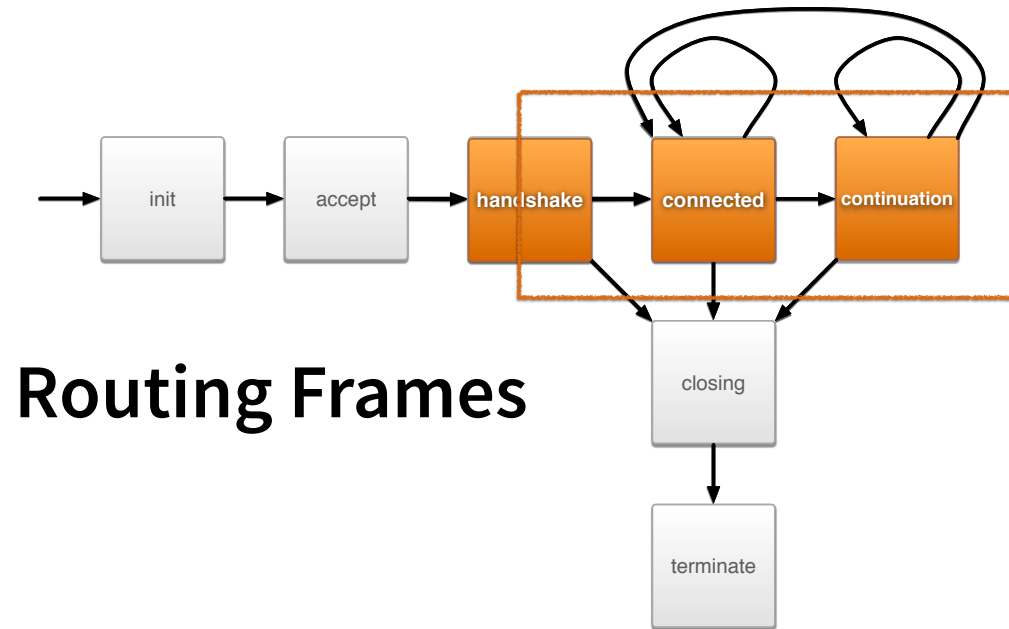
The frame types on the left have connection level operations we need to perform, and the ones on the right are stream level

WINDOW\_UPDATE applies to both

GOAWAY and RST\_STREAM are counterparts at each level

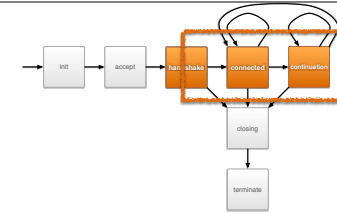
HEADERS, CONTINUATION, PUSH\_PROMISE and DATA frames make up requests and responses.

DATA frames have connection and stream level implications (flow control)



## Routing Frames

There's more parts of the handshake, but that involves using real live HTTP/2 frames. The good news is that the only difference in these states is which frame types we're allowed to process. If we're allowed to process them, they behave the same way. So we'll put the defensive logic in the **StateName/2** callbacks and the processing logic in a function called **route\_frame/2** which we can call from all these states.



```
-spec route_frame(http2_frame(), connection_state())  
-> {next_state,  
    connected | continuation | closing,  
    connection_state(),  
    non_neg_integer()}.  
route_frame(_, State) ->  
    {next_state, connected, State, 0}.
```

This version of the function clause does nothing, but we'll add clauses that do the right things. In =

So, what's this "spec" thing?

It's Erlang's super weak type system which provides compile time type analysis and tells other developers what it is we expect from this function.



Our types are totally weak, they can't possibly fight you

[Empire Strikes] Back  
to the protocol

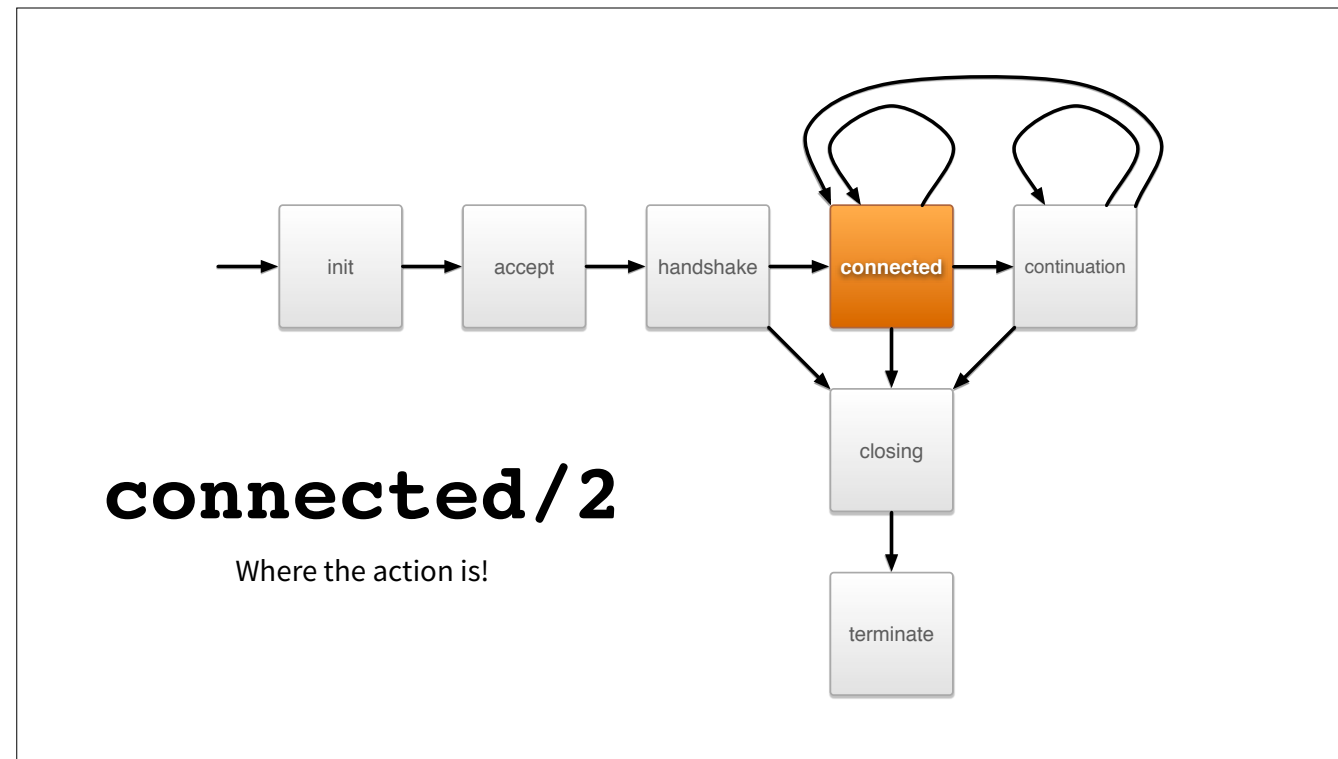
RFC-3P0?



We talked about the settings handshake and how SETTINGS frames are exchanged at the beginning of a connection. I didn't mention how each SETTINGS frame needs to be ACK'd by a response SETTINGS frame with the ACK flag set. This also happens whenever settings are renegotiated.

We didn't have time to discuss this any further during the talk, and I'm hesitant to add this content back in the director's cut of this deck. What you can assume is that there are now **route\_frame/2** clauses that cover this case, and that they can be used both in the **handshake/2** callback and in the **connected/2** callback for the renegotiation.

In the actual talk, I skipped over this, and I'm going to again here :P



At this point we have an open connection and we're just chillin' waiting for frames. As a server, the first frame received is going to be on a new stream id, and it's going to be a HEADERS frame.

NOT TO BE CONFUSED WITH A FRAME HEADER.

A new stream id and a HEADERS frame means this is the beginning of a good old semantic HTTP request, just like mom used to make.

# Frame Types

## Connection Level

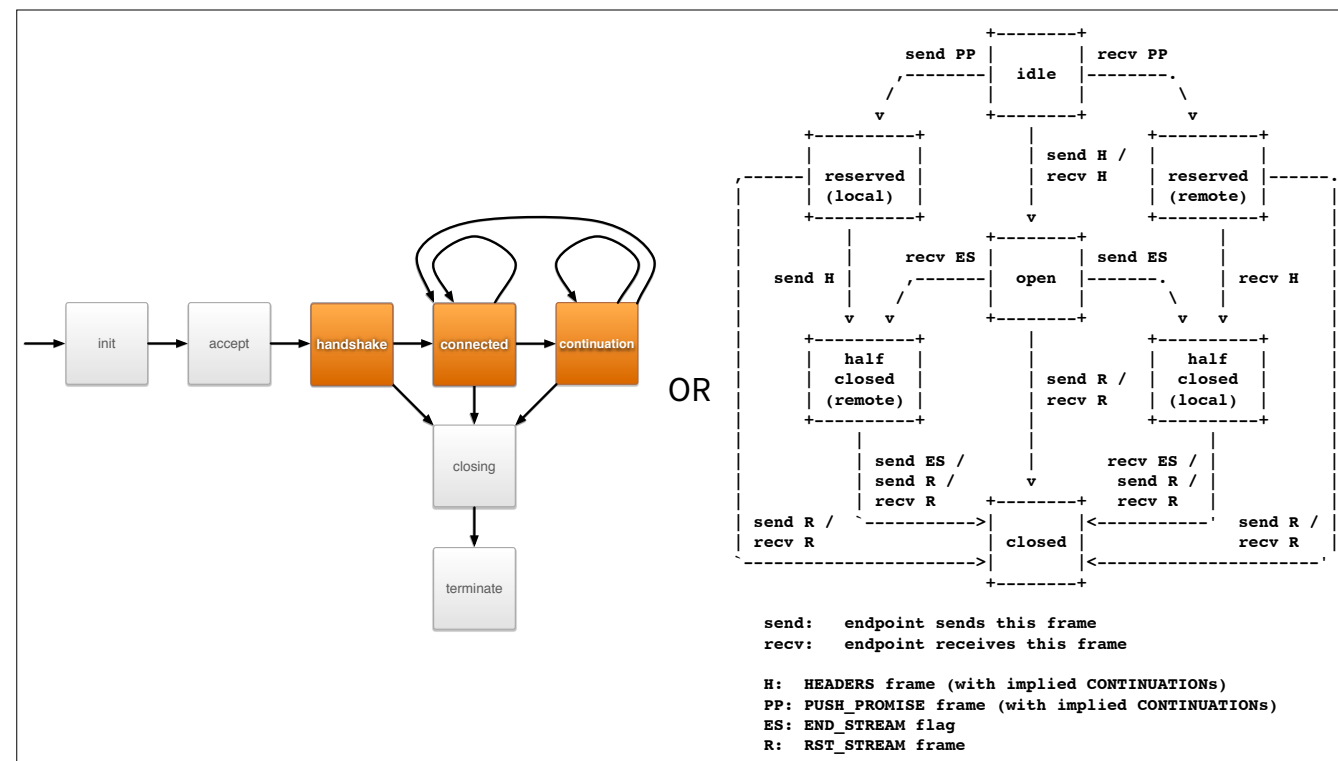
- SETTINGS
- PING
- GOAWAY
- WINDOW\_UPDATE

## Stream Level

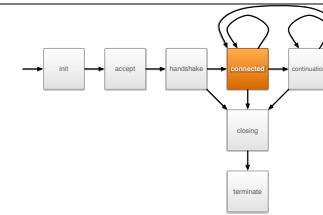
- HEADERS
- PUSH\_PROMISE
- CONTINUATION
- DATA\*
- RST\_STREAM
- WINDOW\_UPDATE
- PRIORITY

It can only ever be one of these, which informs which fsm we're using





The big FSM on the left, or the little one on the right. Don't forget, sometimes implications at both levels like DATA frames. Rather than write another **gen\_fsm** for each stream, we'll maintain each stream's state in the connection state



```

connected(timeout, State=#connection_state{
    socket=Socket
}) ->
{FrameHeader, Payload} = http2_frame:read(Socket),
%% Do stuff, Maybe define "NewState" with an updated
%% value for our remaining flow control credits maybe?
route_frame({FrameHeader, Payload}, State);

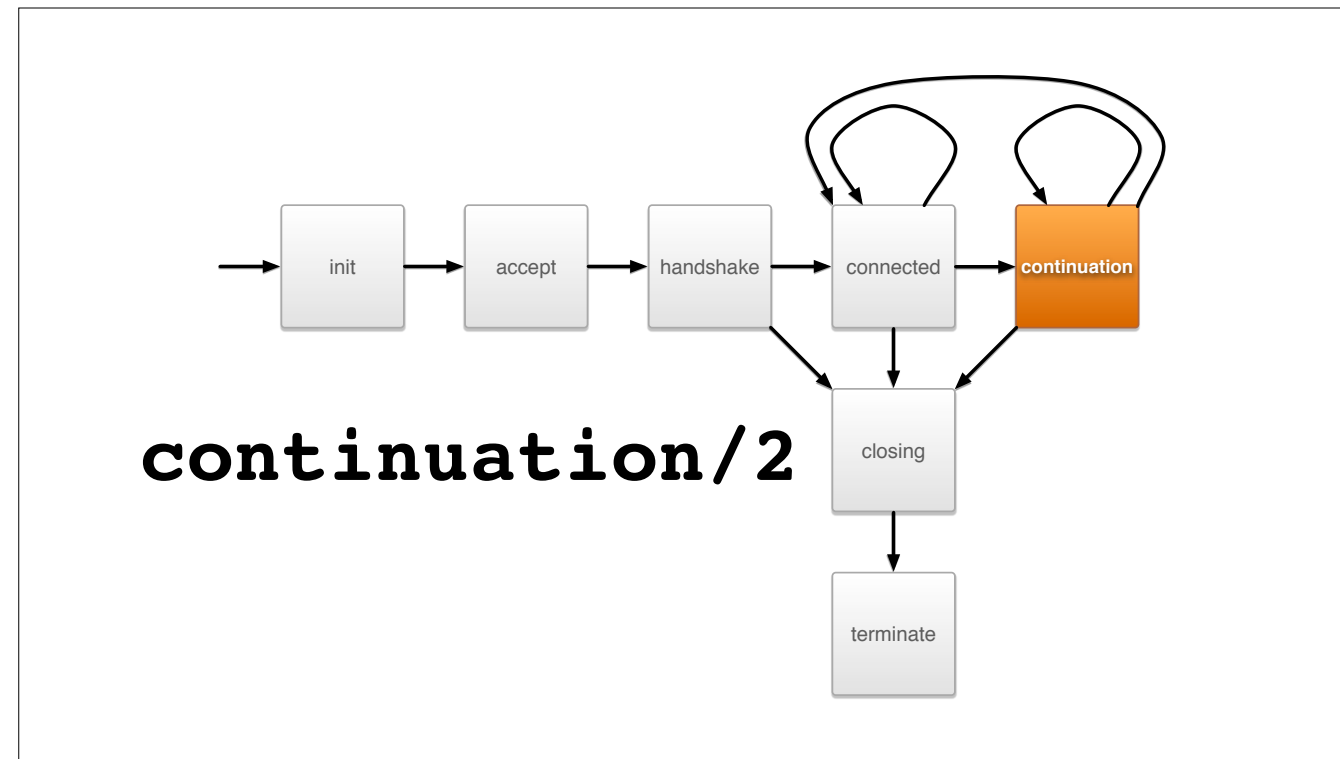
route_frame(_, State) ->
{next_state, connected, State, 0}.

```

The function clause is yet another pattern match on two arguments: a message `timeout` and a bound variable **State**. By bound, I mean we can use **State** anywhere in the function body and it will mean this thing that was passed in. We can't actually change it, but you knew that already.

So we won't change it, but create just a slightly better version of the state, it's like evolutionary state!

We've already got some **route\_frame/2** clauses from the handshake, but we skipped that part, because we don't really need to worry about it for streams



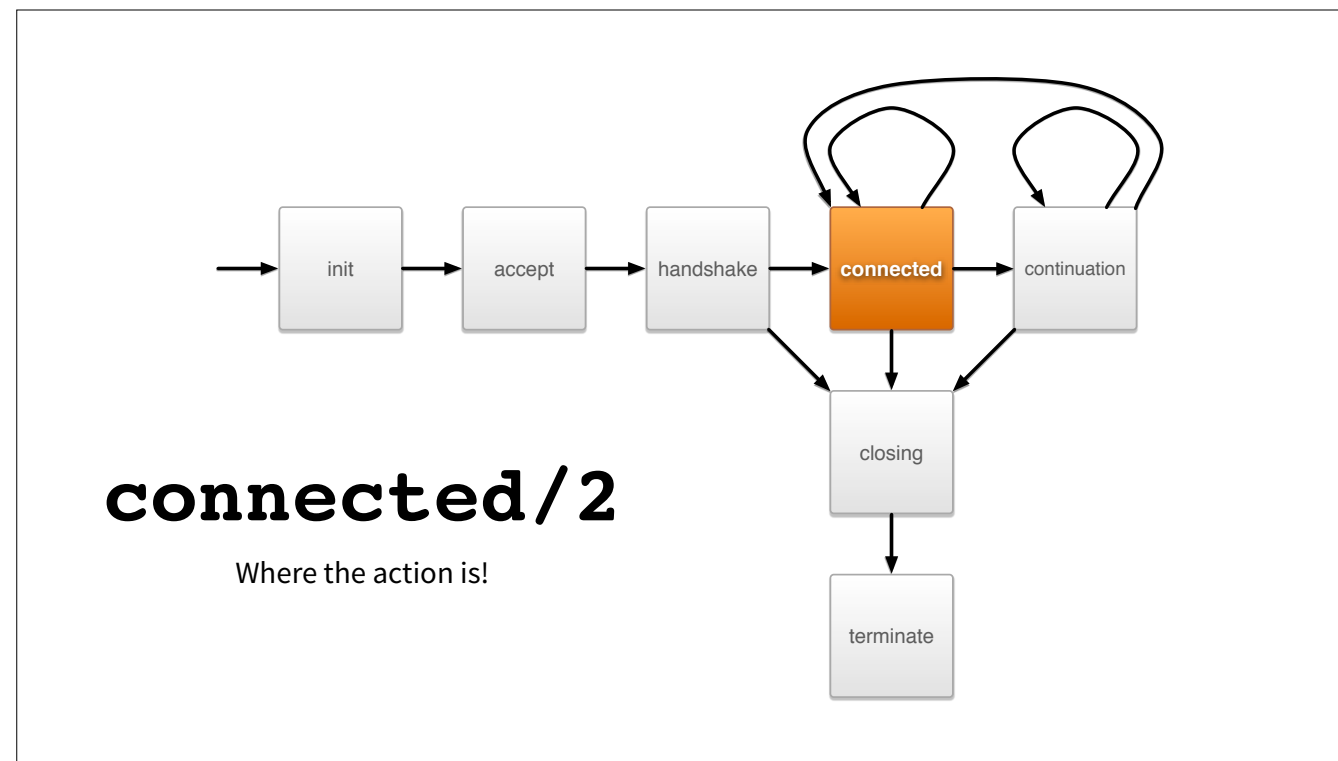
The continuation state is a restricted version of the connected state. But to understand it, you need to understand CONTINUATION frames first. HPACK headers are a binary encoded hunk of data. If that binary hunk is bigger than MAX\_FRAME\_SIZE, it needs to come over in multiple frames. No big deal right?



- If a HEADERS frame without END\_HEADERS flag on Stream N
- Only receive CONTINUATION frames on Stream N until one with the END\_HEADERS flag is received (**continuation** state)
- Then decode them, update the decode context and return to the **connected** state

Since HPACK is always indexing, the indexes are always changing. Since we're using multiplexed streams, it's important that they're processed in order. If they're not, our two copies of the index (client and server side) will get out of sync, and we're screwed.

We solve this with a mutex that's grabbed when a HEADERS frame comes over a stream. The connection will go into lockdown and can only receive CONTINUATION frames on that stream id until the complete set of headers has been sent. The continuation state is effectively a mutex state to solve this.



So back to a connection and it's brand new HTTP request

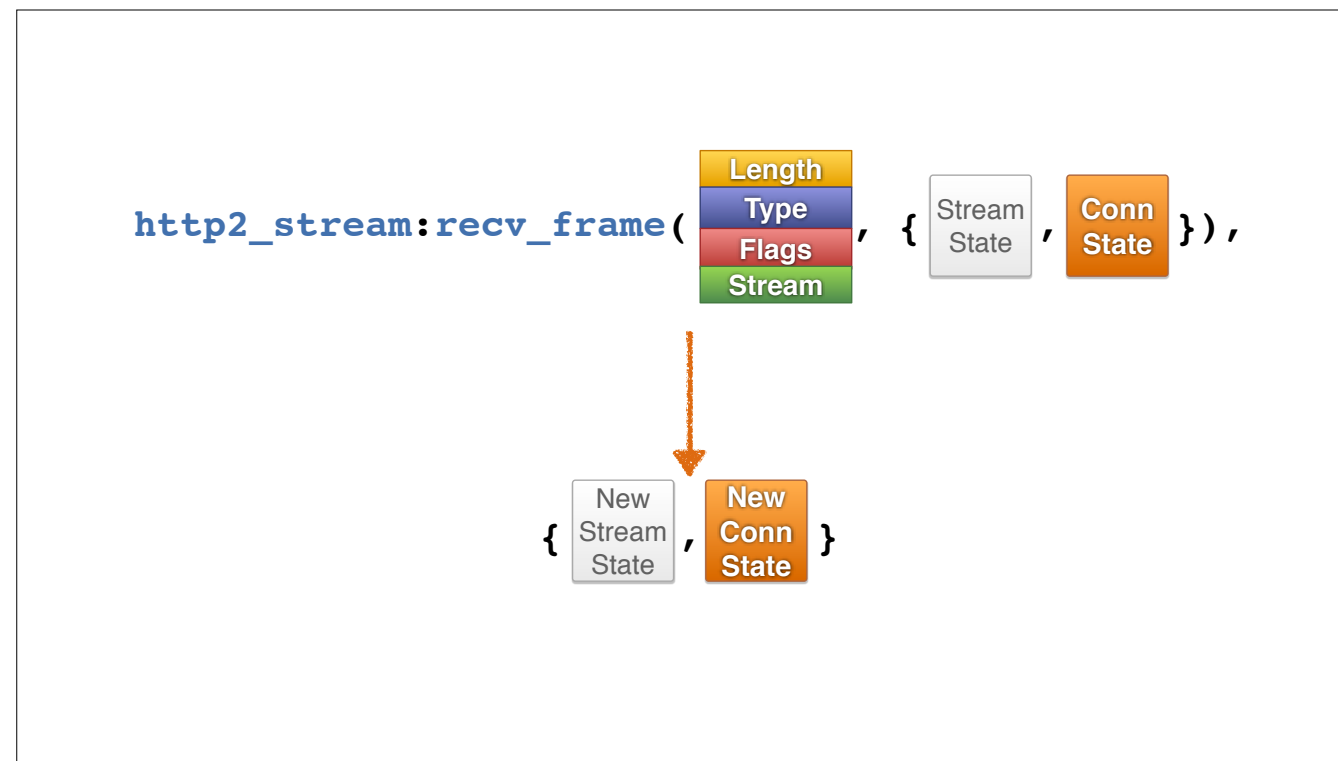
```

route_frame(F={H=#frame_header{
    type=?HEADERS,
    stream_id=StreamId
}, _Payload} = Frame,
    S = #connection_state{
        decode_context=DecodeContext,
        recv_settings=#settings{initial_window_size=RecvWindowSize},
        send_settings=#settings{initial_window_size=SendWindowSize},
        streams=Streams,
        content_handler=Handler
    }) ->
Stream = http2_stream:new(StreamId, {SendWindowSize, RecvWindowSize}),
{NewStream, NewConnectionState}
    = http2_stream:recv_frame(Frame, {Stream, S}),

```

No matter what, when a HEADERS frame comes in, it means create a new stream.

That stream gets initialized and then run through the stream state machine via that **recv\_frame/2** function



**recv\_frame/2** is a stream version of route frame, and knows how to transition that ascii state diagram we talked about. Instead of using a **gen\_fsm** here, which we don't need because we don't need the actor model, we use a referentially transparent function that takes in a frame, the stream state and the connection state. It'll then return a newer, faster, stronger version of those two states. And we can even fold over a list of frames using the two states as an accumulator!

```
NextState = case ?IS_FLAG(H#frame_header.flags, ?FLAG_END_HEADERS) of
  true ->
    connected;
  false ->
    continuation
end,
{next_state, NextState, NewConnectionState#connection_state{
  streams = [{StreamId, NewStream}|Streams],
  continuation_stream_id = StreamId
}, 0}
```

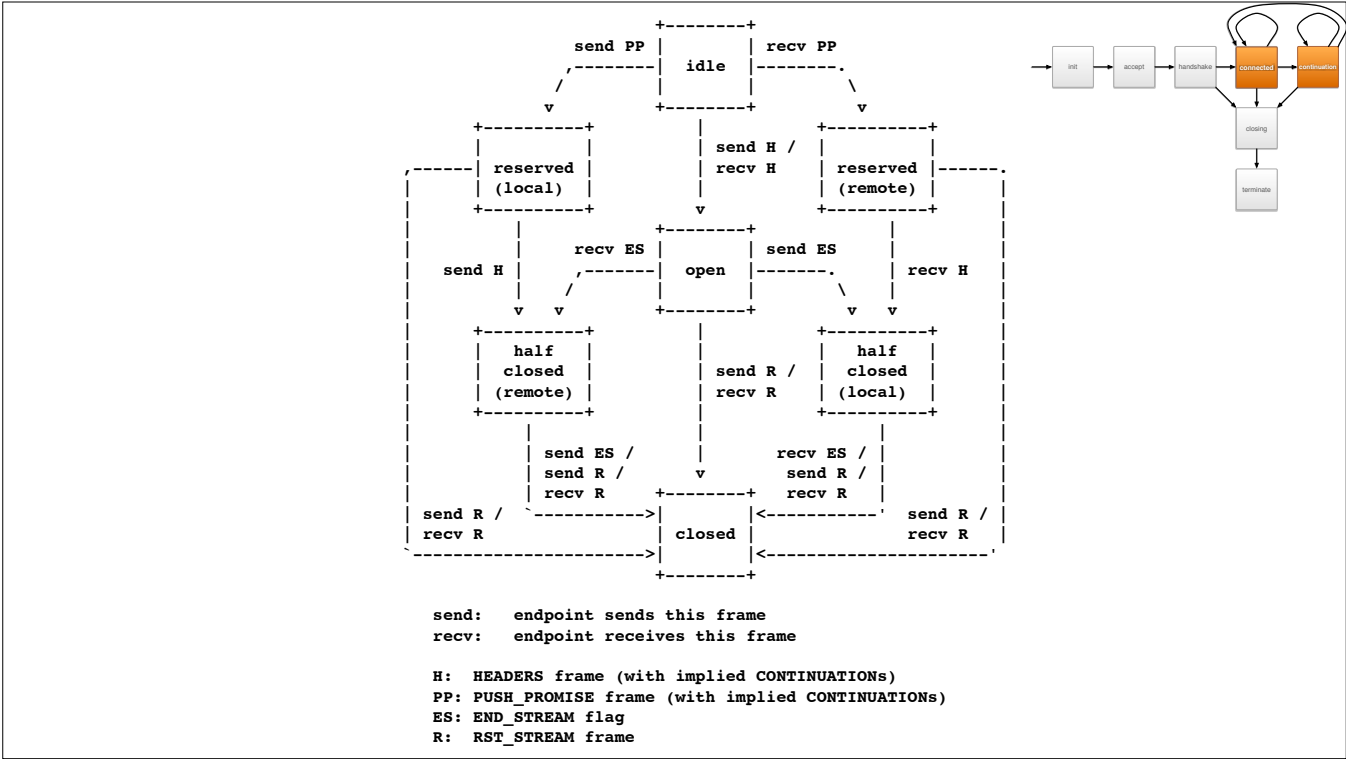
After we do that, we just check for that END\_HEADERS flag to see if we need to take a trip to Muxburg.

Either way, we shove it all in the state and move along to the next callback.

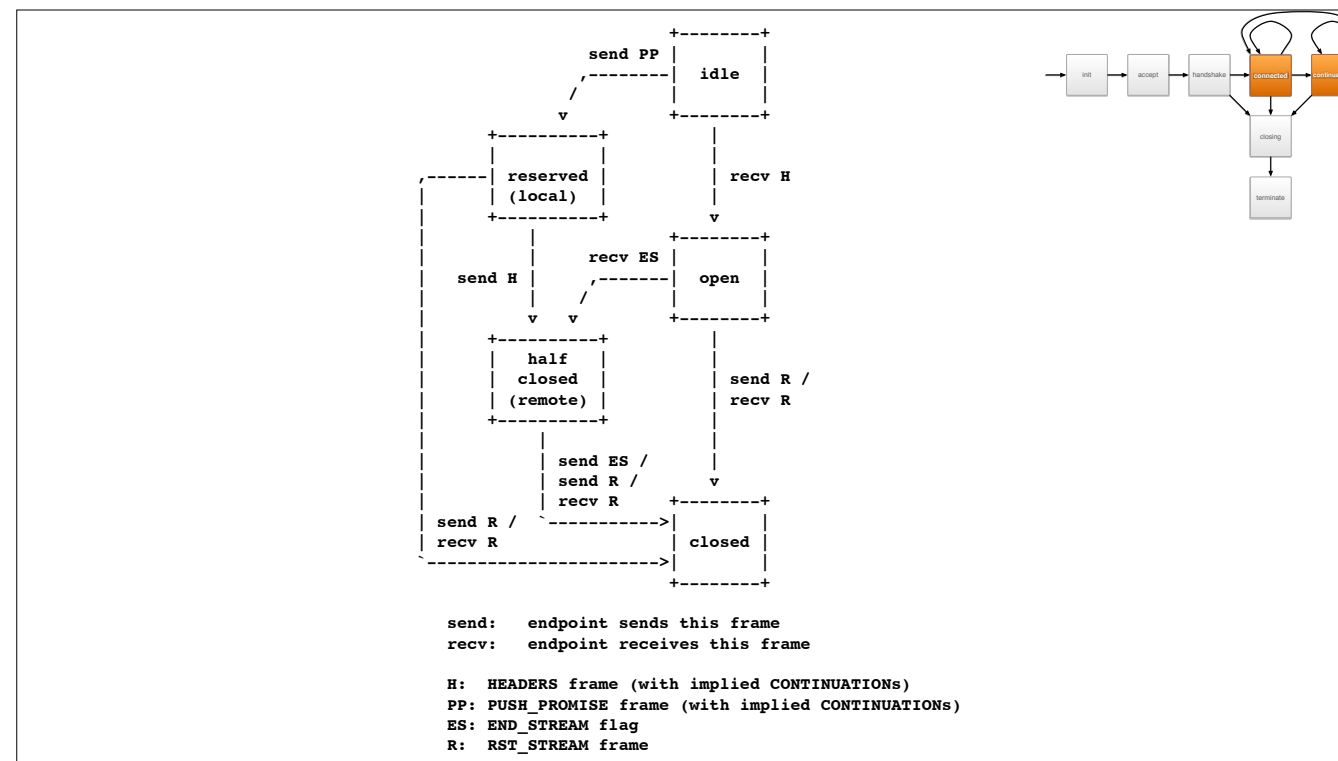


```
-define(IS_FLAG(Flags, Flag), Flags band Flag == Flag).  
-define(NOT_FLAG(Flags, Flag), Flags band Flag /= Flag).
```

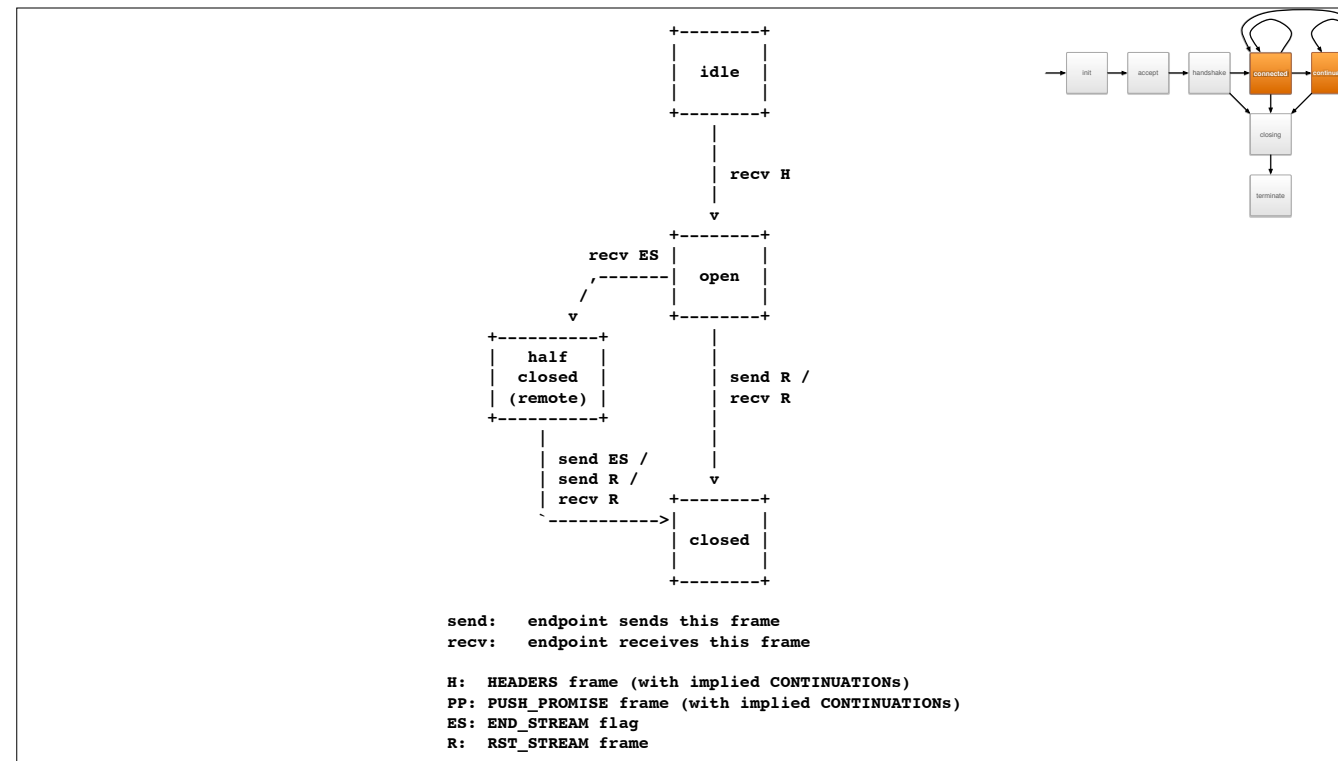
These macros can be used in guards, they're just binary ands.



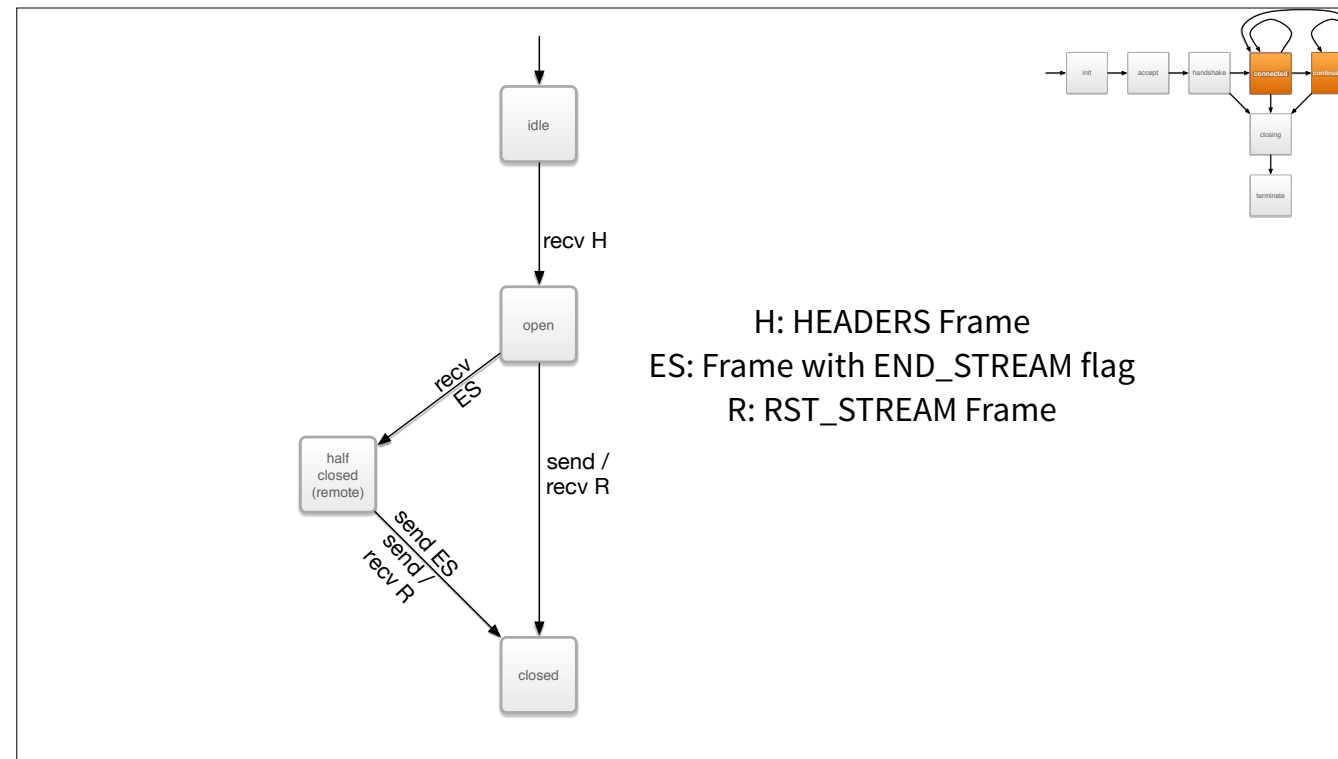
This is straight from the spec and it's confusing because it's trying to show us the client, the server, and push promises.



This is the FSM without the client side.



But we can take out push promises for now and land with this neat little four state, four transition machine



And we can pretty it up too. This is a semantic HTTP request made of HEADERS, CONTINUATIONS, and DATA frames only, broken down into stream states. When we get to **half closed (remote)** it means that the remote connection is done sending on this stream and the request is complete. We then spawn a content handler to process a semantic request



When an Erlang process creates another process, it can also create a link to that process. What that means is that basically if the child process crashes, the parent will get notified. This is key to Erlang's "let it crash" philosophy. We can spin up a content handler in another process, as long as we know which process to send messages back to

```

-spec spawn_handle(
    pid(),
    stream_id(), %% Stream Id
    hpack:headers(), %% Decoded Request Headers
    binary() %% Request Body
) -> pid().
spawn_handle(Pid, StreamId, Headers, ReqBody) ->
    Handler = fun() ->
        handle(Pid, StreamId, Headers, ReqBody)
    end,
    spawn_link(Handler).

-spec handle(
    pid(),
    stream_id(),
    hpack:headers(),
    binary()
) -> ok.

```

half  
closed  
(remote)

send  
send  
recv R

← Define your own!

spawn\_link is a higher order function that takes our **handle/4** function to go run in another process.

Pid (process id) let's know which process our connection **gen\_fsm** is when we want to use that API to send messages back from the spawned process. We also pass in the Stream Id, request headers and request body (if there is one)

```
http2_connection:send_headers(ConnPid, StreamId, ResponseHeaders),  
http2_connection:send_body(ConnPid, StreamId, Data),
```

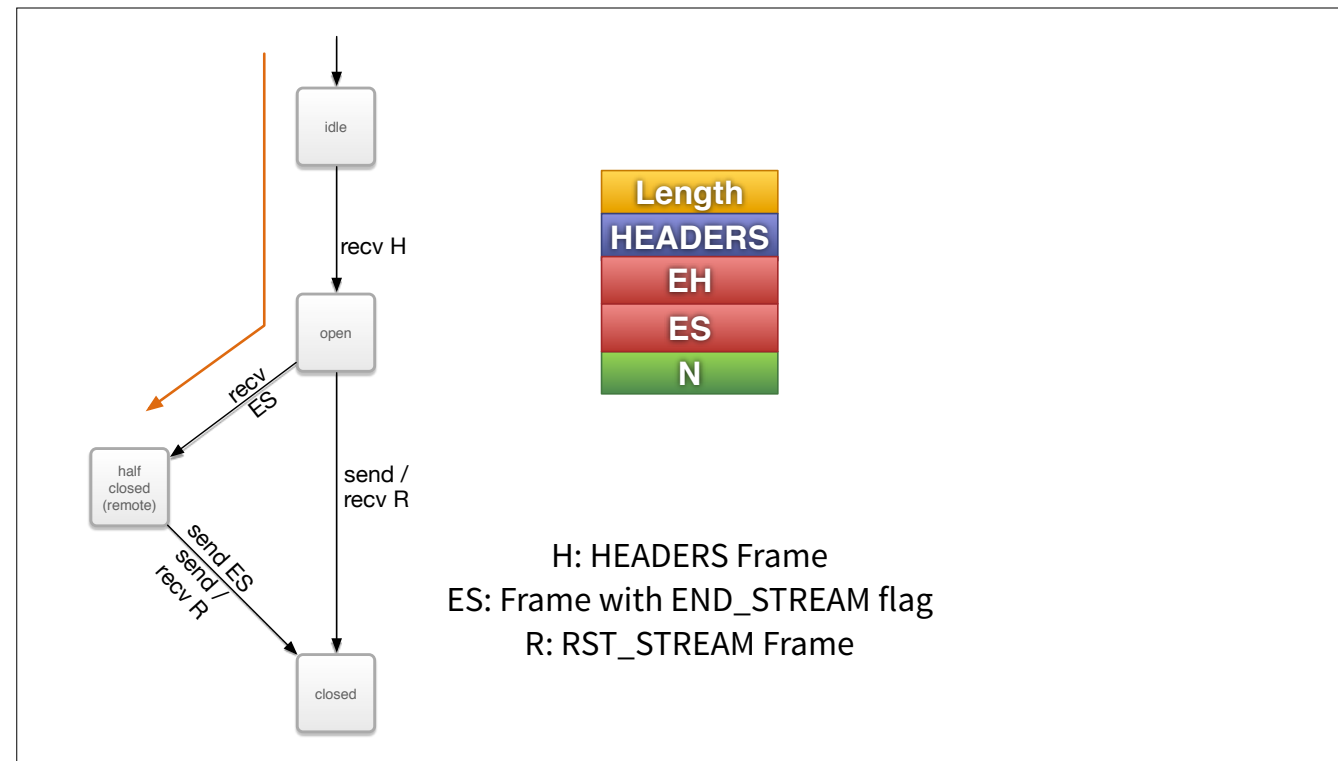
I built a little API for what handlers might need.

Easy right? just send response headers and bodies back over as process messages



## Semantic HTTP Requests through the HTTP/2 Stream State Machine

There's four ways this can happen, I'll show the easiest one first, and then talk about how they all kind of work in the general case.

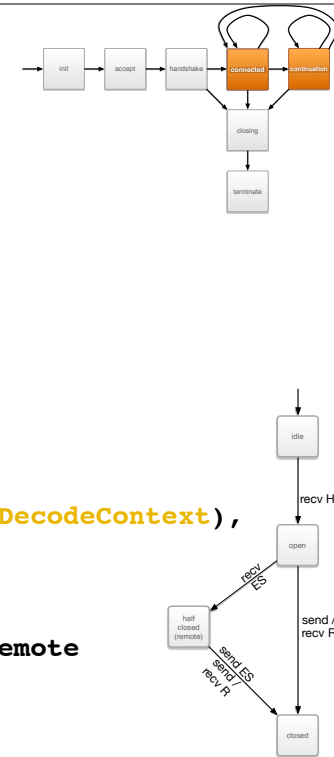


This is the simplest path. This one HEADERS frame with the END\_HEADERS and END\_STREAM flag set is a complete HTTP request. It means the header binary fit in one frame AND there's no request data. We go straight from idle to half closed (remote), decode headers, spawn a process to handle content, and transition to closed when that process is done responding.

```

recv_frame(F={#frame_header{
    flags=Flags,
    type=?HEADERS
}, _Payload},
{Stream=#stream_state{
    state=idle
},
Connection=#connection_state{
    decode_context=DecodeContext,
    content_handler=Handler
}})
when ?IS_FLAG(Flags, ?FLAG_END_STREAM),
    ?IS_FLAG(Flags, ?FLAG_END_HEADERS) ->
    HeadersBin = http2_frame_headers:from_frames([F]),
    {Headers, NewDecodeContext} = hpack:decode(HeadersBin, DecodeContext),
    Handler:spawn_handle(self(), StreamId, Headers, <<>>),
    {Stream#stream_state{request_headers=Headers},
    Connection#connection_state{
        decode_context=NewDecodeContext, state=half_closed_remote
    }
    };

```

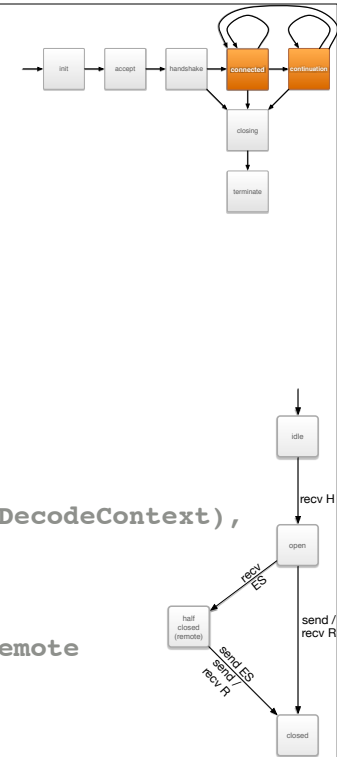


Here's the code for the simplest path. We make sure we're working with a HEADERS frame

```

recv_frame(F={#frame_header{
    flags=Flags,
    type=?HEADERS
}, _Payload},
{Stream=#stream_state{
    state=idle
},
Connection=#connection_state{
    decode_context=DecodeContext,
    content_handler=Handler
}})
when ?IS_FLAG(Flags, ?FLAG_END_STREAM),
    ?IS_FLAG(Flags, ?FLAG_END_HEADERS) ->
    HeadersBin = http2_frame_headers:from_frames([F]),
    {Headers, NewDecodeContext} = hpack:decode(HeadersBin, DecodeContext),
    Handler:spawn_handle(self(), StreamId, Headers, <<>>),
    {Stream#stream_state{request_headers=Headers},
    Connection#connection_state{
        decode_context=NewDecodeContext, state=half_closed_remote
    }
    }
};

```



Our guard here tests to make sure this HEADERS frame has END\_HEADERS and END\_STREAM

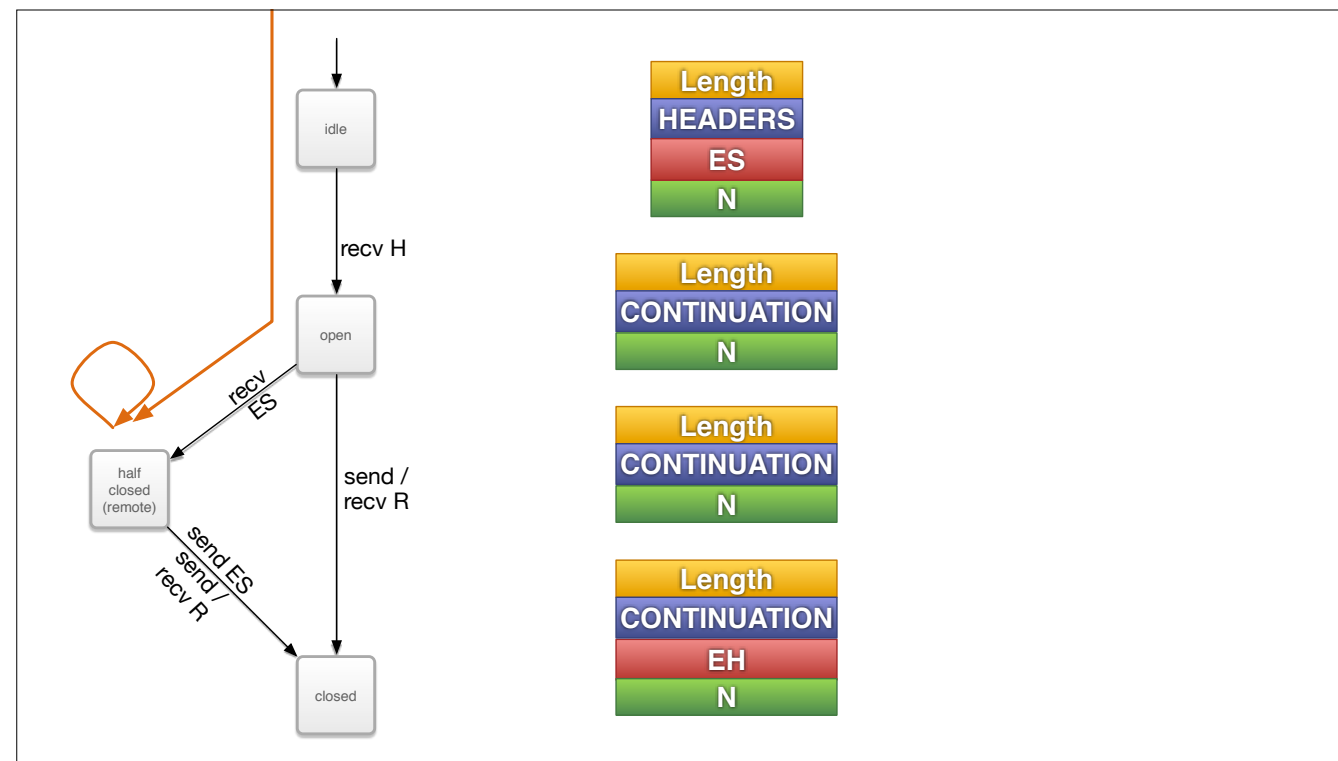


The first highlighted line here turns a list of HEADERS frames into a Headers Binary. In this case, that list only has a single frame.

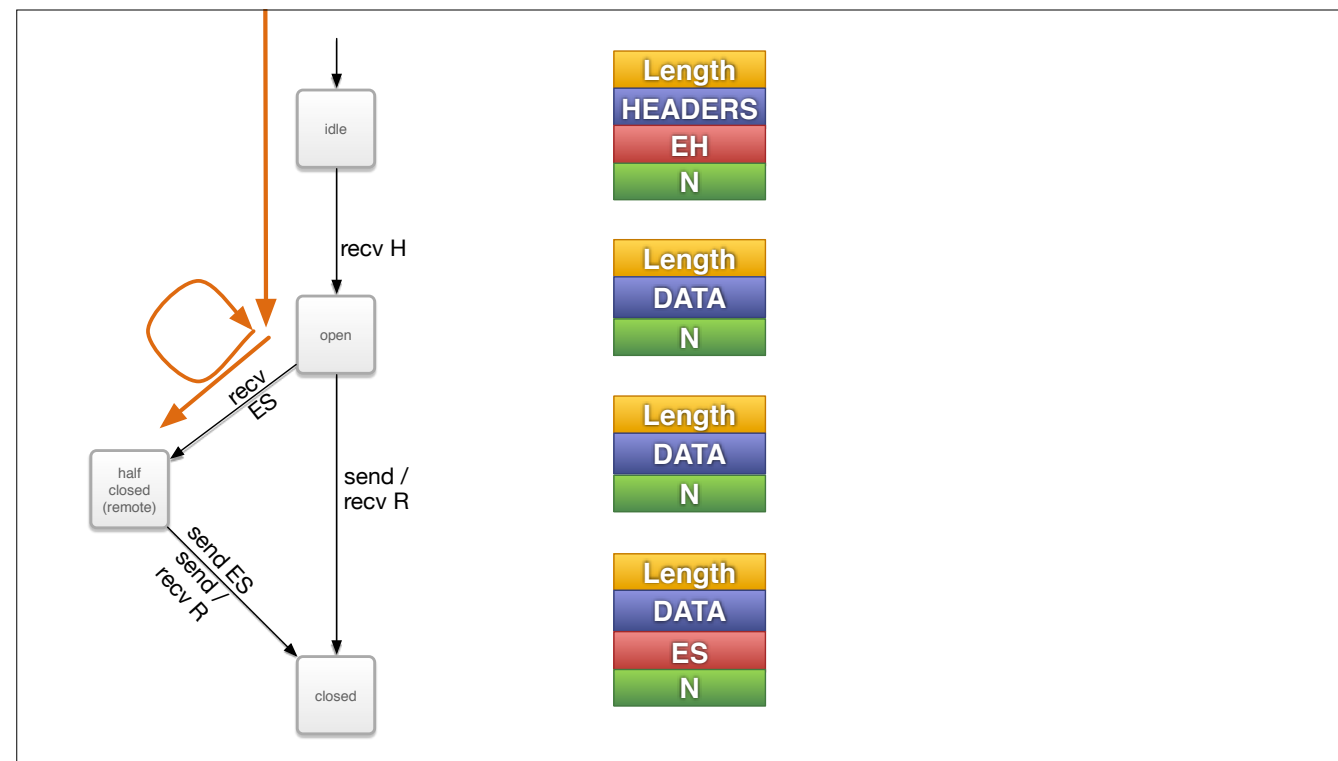
In the second highlighted line decodes that binary using the HPACK decode context. We return a new context, that's the new header table!

The last highlighted line is spawning the content handler. In this case we're serving up files as DATA frames, but we could and probably will write a webmachine handler that could be used here as well.

Since we know this is an END\_HEADERS, we don't even need to transition into the continuation state. We don't need the mutex since we've already met the target condition! We'll go set the NextState value back at the connection level



[Director's Cut] This is the second path we could take:  
Idle receives HEADERS with END\_STREAM, no END\_HEADERS,  
transition to half\_closed\_remote,  
wait for continuations until END HEADERS,  
then the request ends and we spawn the handler, etc...

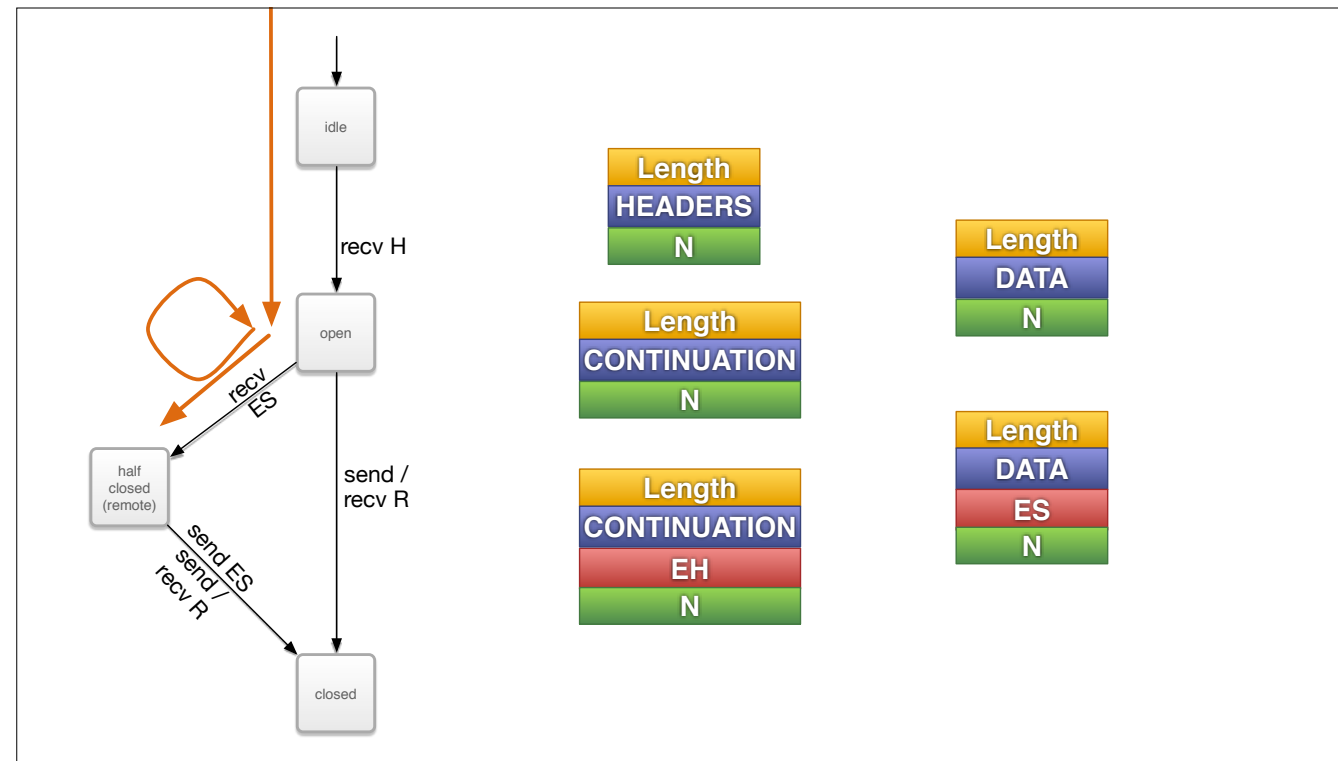


[Director's Cut] Path 3:

idle receives HEADERS with END\_HEADERS, no END\_STREAM.

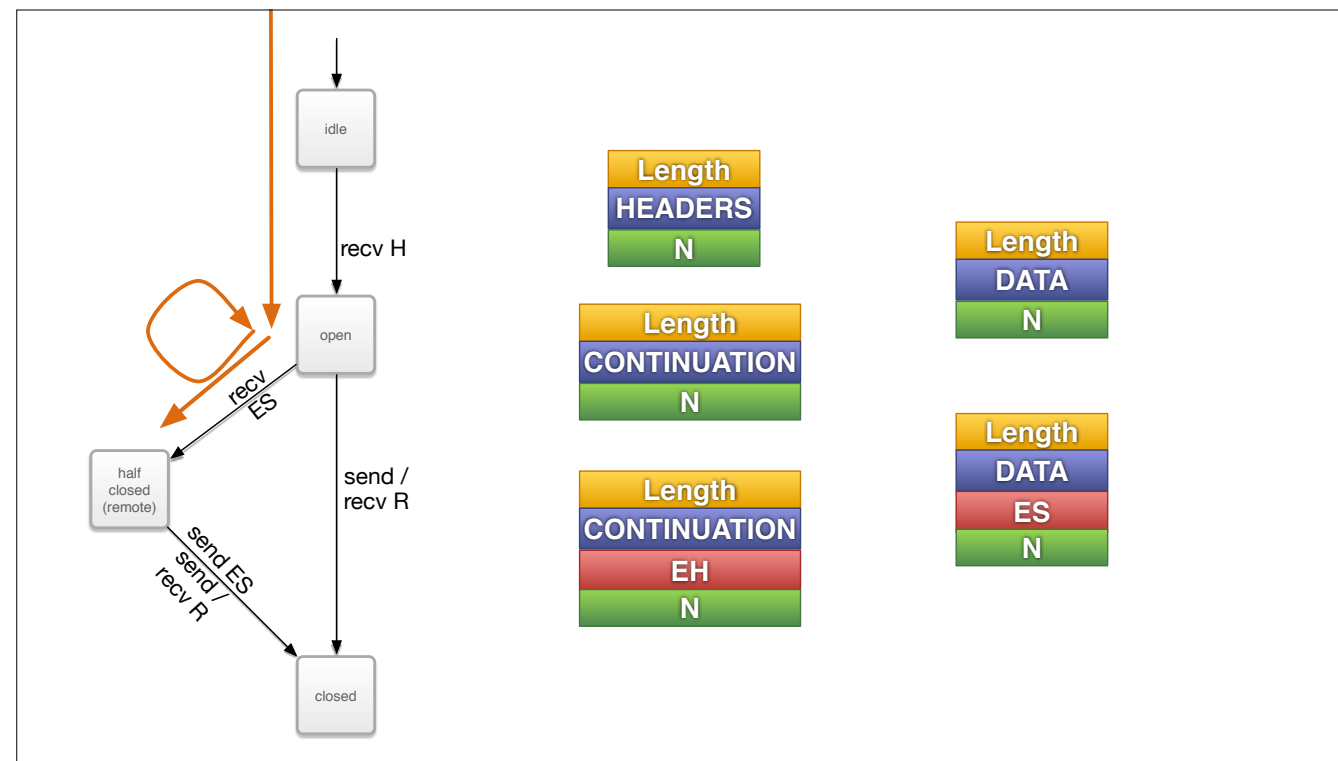
transition to open and wait for DATA frames until one comes with END\_STREAM

End request, spawn handler, etc...



[Director's Cut] Path 4: The most complex  
idle receives HEADERS, no END\_STREAM or END\_HEADERS  
transition into open,  
expect continuations until one shows up with an END\_HEADERS,  
then expect DATA frames until one shows up with an END\_STREAM  
then transition to half closed remote, ending the request. Spawn the handler, etc...





All the paths pretty much work like this:

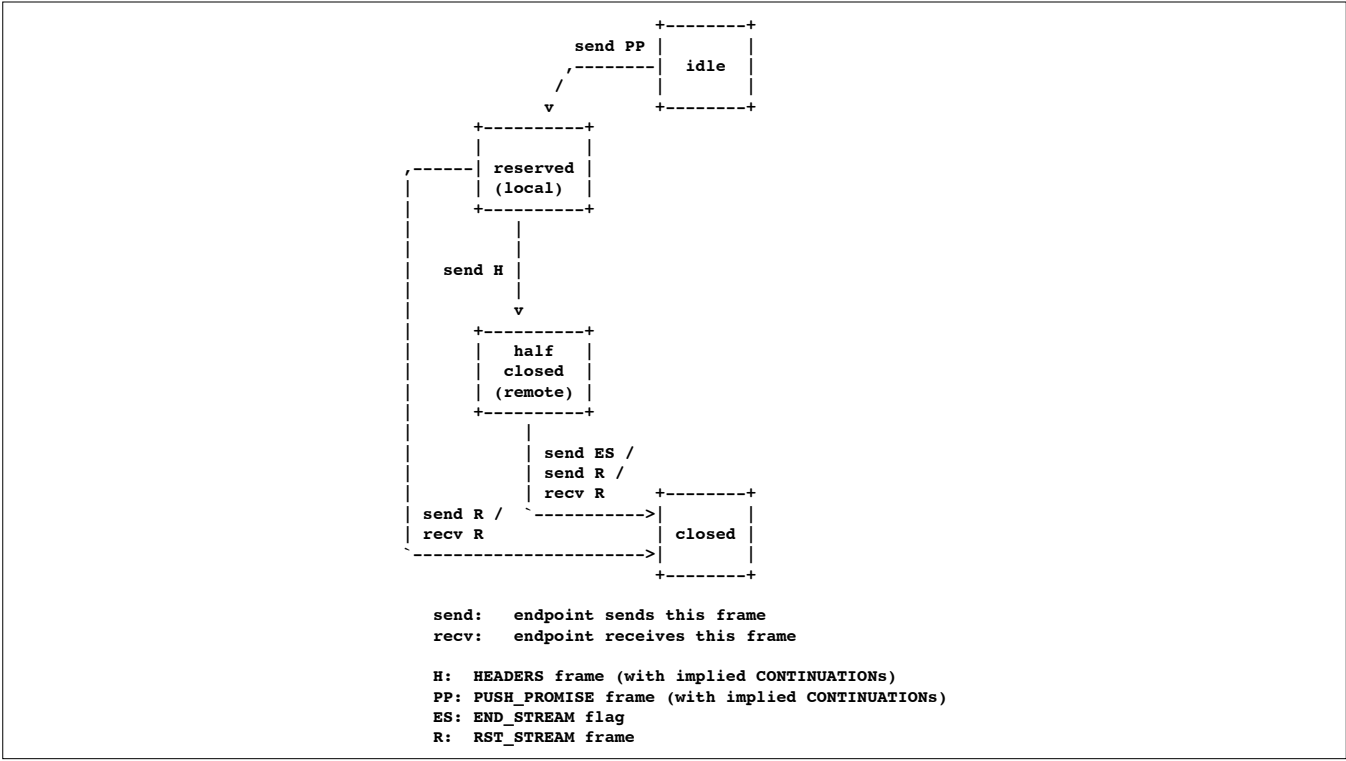
A Headers frame moves us into the open state, where we hang out receiving the rest of the request. Could be header continuation frames, could be data frames. We'll know when it's over when we get the **END\_STREAM**.

Half closed remote is where the action is!

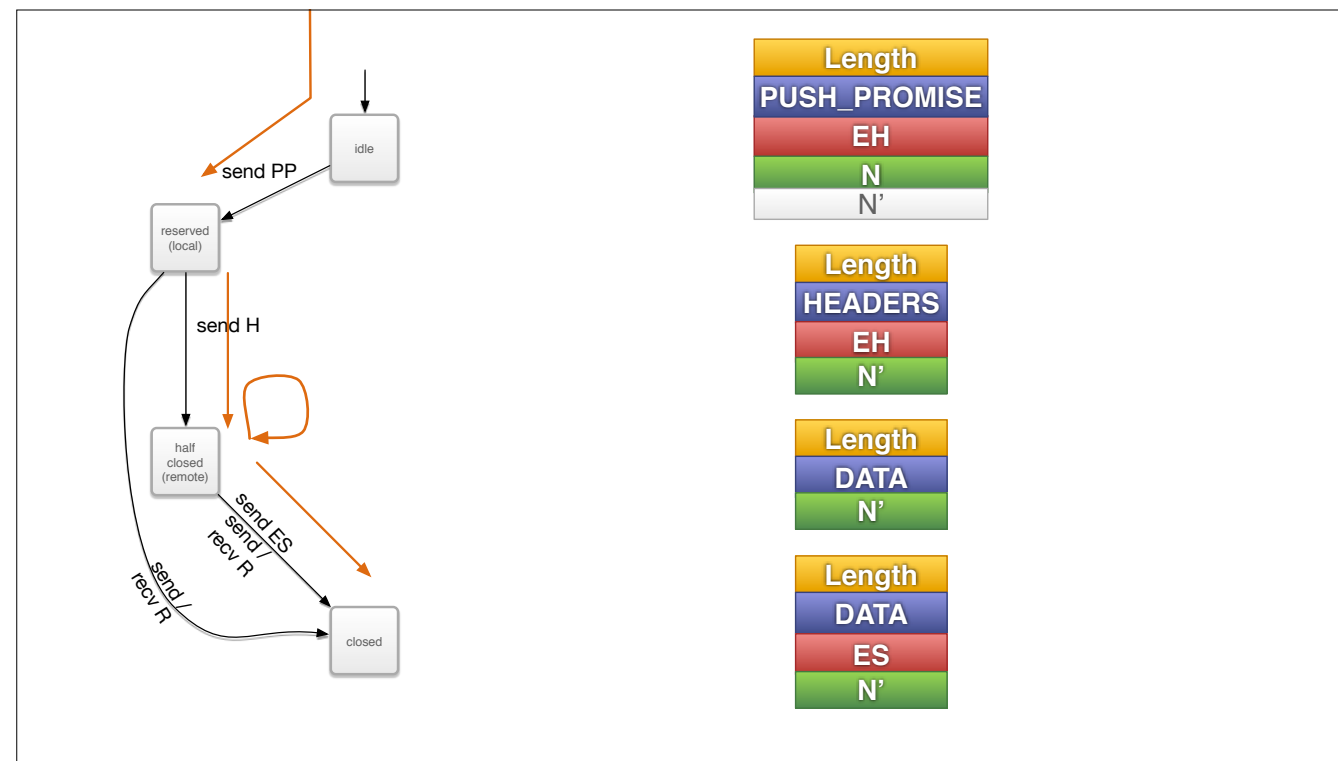
## Sending a PUSH PROMISE

```
NewStreamId = http2_connection:new_stream(ConnPid),  
http2_connection:send_promise(ConnPid, StreamId, NewStreamId, PHeaders),  
spawn_handle(ConnPid, NewStreamId, PHeaders, <<>>),
```

Here are the additional API calls I need to send a push promise



Here's the state machine with open removed, to illustrate the life of a PUSH\_PROMISE



This is all sent from the server, NO REQUEST. The PP frame is sent on the N stream, but it's payload tells the client to reserve the N' stream for promised data

# What went well?

- Binary Pattern Matching
- Spawn\_link content handler
- OTP Supervisor / Gen\_FSM
- Erlang's socket library

Erlang's spawn\_link function made adding concurrent stream handling something I was able to add in a couple of hours.

Thank You SO MUCH!

I learned so much doing this, and there's so much more to do, and so much I haven't talked about. Hopefully I made HTTP/2 and/or Erlang less intimidating... FOR YOU.