Ranch User Guide

Contents

I Interface 1

1 Introduction 2

1.1 Prerequisites . . . 2
1.2 Supported platforms . . . 2
1.3 Versioning . . . 2

2 Listeners 3

2.1 Starting a listener . . . 3
2.2 Stopping a listener . . . 4
2.3 Suspending and resuming a listener . . . 4
2.4 Default transport options . . . 5
2.5 Listening on a random port . . . 5
2.6 Listening on privileged ports . . . 5
2.7 Accepting connections on an existing socket . . . 5
2.8 Limiting the number of concurrent connections . . . 5
2.9 Customizing the number of acceptor processes . . . 6
2.10 When running out of file descriptors . . . 6
2.11 Using a supervisor for connection processes . . . 7
2.12 Upgrading . . . 7
2.13 Changing transport options . . . 7
2.14 Obtaining information about listeners . . . 8

3 Transports 9

3.1 TCP transport . . . 9
3.2 SSL transport . . . 9
3.3 Sending and receiving data . . . 9
3.4 Sending files . . . 10
3.5 Upgrading a TCP socket to SSL . . . 11
3.6 Writing a transport handler . . . 11

4 Protocols **12**
4.1 Writing a protocol handler . . . 12
4.2 Using gen_statem . . . 13

5 Embedded mode **14**
5.1 Embedding . . . 14

II How to **15**

6 Writing parsers **16**
6.1 Parsing text . . . 16
6.2 Parsing binary . . . 17

7 SSL client authentication **18**
7.1 Purpose . . . 18
7.2 Obtaining client certificates . . . 18
7.3 Transport configuration . . . 19
7.4 Authentication . . . 19

III Advanced **20**

8 Internals **21**
8.1 Architecture . . . 21
8.2 Number of acceptors . . . 21
8.3 Platform-specific TCP features . . . 22

IV Additional information **23**

A Upcoming changes in Ranch 2.0 **24**

B Changes since Ranch 1.8 **25**
B.1 Ranch 1.8.1 . . . 25

C Migrating from Ranch 1.7 to 1.8 **26**
C.1 Bugs fixed . . . 26

D Migrating from Ranch 1.6 to 1.7 **27**
D.1 Features added . . . 27
D.2 New functions . . . 27
D.3 Bugs fixed . . . 27

E Migrating from Ranch 1.5 to 1.6 **28**

E.1 Features added . . . 28

E.2 Experimental features added . . . 28

E.3 Changed behaviors . . . 28

E.4 New functions . . . 29

E.5 Bugs fixed . . . 29

F Migrating from Ranch 1.x **30**

F.1 1.5.0 . . . 30

F.2 1.4.0 . . . 30

F.3 1.3.0 . . . 30

F.4 1.2.1 . . . 31

F.5 1.2.0 . . . 31

F.6 1.1.0 . . . 31

F.7 1.0.0 . . . 31

Part I

Interface

Chapter 1

Introduction

Ranch is a socket acceptor pool for TCP protocols.

Ranch aims to provide everything you need to accept TCP connections with a small code base and low latency while being easy to use directly as an application or to embed into your own.

1.1 Prerequisites

It is assumed the developer already knows Erlang and has some experience with socket programming and TCP protocols.

1.2 Supported platforms

Ranch is tested and supported on Linux, FreeBSD, OSX and Windows.

Ranch is developed for Erlang/OTP 19+.

Ranch may be compiled on earlier Erlang versions with small source code modifications but there is no guarantee that it will work as expected.

1.3 Versioning

Ranch uses Semantic Versioning 2.0.0

Chapter 2

Listeners

A listener is a set of processes whose role is to listen on a port for new connections. It manages a pool of acceptor processes, each of them indefinitely accepting connections. When it does, it starts a new process executing the protocol handler code. All the socket programming is abstracted through the use of transport handlers.

The listener takes care of supervising all the acceptor and connection processes, allowing developers to focus on building their application.

2.1 Starting a listener

Ranch does nothing by default. It is up to the application developer to request that Ranch listens for connections.

A listener can be started and stopped at will.

When starting a listener, a number of different settings are required:

- A name to identify it locally and be able to interact with it.
- The number of acceptors in the pool.
- A transport handler and its associated options.
- A protocol handler and its associated options.

Ranch includes both TCP and SSL transport handlers, respectively `ranch_tcp` and `ranch_ssl`.

A listener can be started by calling the `ranch:start_listener/5` function. Before doing so however, you must ensure that the `ranch` application is started.

Starting the Ranch application

```
ok = application:start(ranch).
```

You are then ready to start a listener. Let's call it `tcp_echo`. It will have a pool of 100 acceptors, use a TCP transport and forward connections to the `echo_protocol` handler.

Starting a listener for TCP connections on port 5555

```
{ok, _} = ranch:start_listener(tcp_echo,
    ranch_tcp, [{port, 5555}],
    echo_protocol, []
).
```

You can try this out by compiling and running the `tcp_echo` example in the examples directory. To do so, open a shell in the *examples/tcp_echo/* directory and run the following command:

Building and starting a Ranch example

```
$ make run
```

You can then connect to it using telnet and see the echo server reply everything you send to it. Then when you're done testing, you can use the `Ctrl+]` key to escape to the telnet command line and type `quit` to exit.

Connecting to the example listener with telnet

```
$ telnet localhost 5555
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello!
Hello!
It works!
It works!
^]

telnet> quit
Connection closed.
```

2.2 Stopping a listener

All you need to stop a Ranch listener is to call the `ranch:stop_listener/1` function with the listener's name as argument. In the previous section we started the listener named `tcp_echo`. We can now stop it.

Stopping a listener

```
ranch:stop_listener(tcp_echo).
```

2.3 Suspending and resuming a listener

Listeners can be suspended and resumed by calling `ranch:suspend_listener/1` and `ranch:resume_listener/1`, respectively, with the name of the listener as argument.

Suspending a listener will cause it to stop listening and not accept new connections, but existing connection processes will not be stopped.

Suspending a listener

```
ranch:suspend_listener(tcp_echo).
```

Resuming a listener will cause it to start listening and accept new connections again. It is worth mentioning, however, that if the listener is configured to listen on a random port, it will listen on a different port than before it was suspended.

Resuming a listener

```
ranch:resume_listener(tcp_echo).
```

Whether a listener is currently running or suspended can be queried by calling `ranch:get_status/1` with the listener name as argument.

2.4 Default transport options

By default the socket will be set to return `binary` data, with the options `{active, false}`, `{packet, raw}`, `{reuseaddr, true}` set. These values can't be overriden when starting the listener, but they can be overriden using `Transport:setopts/2` in the protocol.

It will also set `{backlog, 1024}` and `{nodelay, true}`, which can be overriden at listener startup.

2.5 Listening on a random port

You do not have to specify a specific port to listen on. If you give the port number 0, or if you omit the port number entirely, Ranch will start listening on a random port.

You can retrieve this port number by calling `ranch:get_port/1`. The argument is the name of the listener you gave in `ranch:start_listener/5`.

Starting a listener for TCP connections on a random port

```
{ok, _} = ranch:start_listener(tcp_echo,
    ranch_tcp, [{port, 0}],
    echo_protocol, []
).
Port = ranch:get_port(tcp_echo).
```

2.6 Listening on privileged ports

Some systems limit access to ports below 1024 for security reasons. This can easily be identified by an `{error, eacces}` error when trying to open a listening socket on such a port.

The methods for listening on privileged ports vary between systems, please refer to your system's documentation for more information.

We recommend the use of port rewriting for systems with a single server, and load balancing for systems with multiple servers. Documenting these solutions is however out of the scope of this guide.

2.7 Accepting connections on an existing socket

If you want to accept connections on an existing socket, you can use the `socket` transport option, which should just be the relevant data returned from the connect function for the transport or the underlying socket library (`gen_tcp:connect`, `ssl:connect`). The accept function will then be called on the passed in socket. You should connect the socket in `{active, false}` mode, as well.

2.8 Limiting the number of concurrent connections

The `max_connections` transport option allows you to limit the number of concurrent connections. It defaults to 1024. Its purpose is to prevent your system from being overloaded and ensuring all the connections are handled optimally.

Customizing the maximum number of concurrent connections

```
{ok, _} = ranch:start_listener(tcp_echo,
    ranch_tcp, [{port, 5555}, {max_connections, 100}],
    echo_protocol, []
).
```

You can disable this limit by setting its value to the atom `infinity`.

Disabling the limit for the number of connections

```
{ok, _} = ranch:start_listener(tcp_echo,
    ranch_tcp, [{port, 5555}, {max_connections, infinity}],
    echo_protocol, []
).
```

The maximum number of connections is a soft limit. In practice, it can reach `max_connections` + the number of acceptors.

When the maximum number of connections is reached, Ranch will stop accepting connections. This will not result in further connections being rejected, as the kernel option allows queueing incoming connections. The size of this queue is determined by the `backlog` option and defaults to 1024. Ranch does not know about the number of connections that are in the backlog.

You may not always want connections to be counted when checking for `max_connections`. For example you might have a protocol where both short-lived and long-lived connections are possible. If the long-lived connections are mostly waiting for messages, then they don't consume much resources and can safely be removed from the count.

To remove the connection from the count, you must call the `ranch:remove_connection/1` from within the connection process, with the name of the listener as the only argument.

Removing a connection from the count of connections

```
ranch:remove_connection(Ref).
```

As seen in the chapter covering protocols, this pid is received as the first argument of the protocol's `start_link/4` callback.

You can modify the `max_connections` value on a running listener by using the `ranch:set_max_connections/2` function, with the name of the listener as first argument and the new value as the second.

Upgrading the maximum number of connections

```
ranch:set_max_connections(tcp_echo, MaxConns).
```

The change will occur immediately.

2.9 Customizing the number of acceptor processes

By default Ranch will use 10 acceptor processes. Their role is to accept connections and spawn a connection process for every new connection.

This number can be tweaked to improve performance. A good number is typically between 10 or 100 acceptors. You must measure to find the best value for your application.

Specifying a custom number of acceptor processes

```
{ok, _} = ranch:start_listener(tcp_echo,
    ranch_tcp, [{port, 5555}, {num_acceptors, 42}],
    echo_protocol, []
).
```

2.10 When running out of file descriptors

Operating systems have limits on the number of sockets which can be opened by applications. When this maximum is reached the listener can no longer accept new connections. The accept rate of the listener will be automatically reduced, and a warning message will be logged.

```
=ERROR REPORT==== 13-Jan-2016::12:24:38 ===
Ranch acceptor reducing accept rate: out of file descriptors
```

If you notice messages like this you should increase the number of file-descriptors which can be opened by your application. How this should be done is operating-system dependent. Please consult the documentation of your operating system.

2.11 Using a supervisor for connection processes

Ranch allows you to define the type of process that will be used for the connection processes. By default it expects a `worker`. When the `connection_type` configuration value is set to `supervisor`, Ranch will consider that the connection process it manages is a supervisor and will reflect that in its supervision tree.

Connection processes of type `supervisor` can either handle the socket directly or through one of their children. In the latter case the start function for the connection process must return two pids: the pid of the supervisor you created (that will be supervised) and the pid of the protocol handling process (that will receive the socket).

Instead of returning `{ok, ConnPid}`, simply return `{ok, SupPid, ConnPid}`.

It is very important that the connection process be created under the supervisor process so that everything works as intended. If not, you will most likely experience issues when the supervised process is stopped.

2.12 Upgrading

Ranch allows you to upgrade the protocol options. This takes effect immediately and for all subsequent connections.

To upgrade the protocol options, call `ranch:set_protocol_options/2` with the name of the listener as first argument and the new options as the second.

Upgrading the protocol options

```
ranch:set_protocol_options(tcp_echo, NewOpts).
```

All future connections will use the new options.

You can also retrieve the current options similarly by calling `ranch:get_protocol_options/1`.

Retrieving the current protocol options

```
Opts = ranch:get_protocol_options(tcp_echo).
```

2.13 Changing transport options

Ranch allows you to change the transport options of a listener, for example to make it listen on a different port.

To change transport options, the listener has to be suspended first. Then you are allowed to change the transport options by calling `ranch:set_transport_options/2` with the listener name and the new transport options as arguments. After that, you can resume the listener.

Changing the transport options

```
ranch:set_transport_options(tcp_echo, NewOpts).
```

You can retrieve the current transport options by calling `ranch:get_transport_options/1`.

Retrieving the current transport options

```
Opts = ranch:get_transport_options(tcp_echo).
```

2.14 Obtaining information about listeners

Ranch provides two functions for retrieving information about the listeners, for reporting and diagnostic purposes.

The `ranch:info/0` function will return detailed information about all listeners.

Retrieving detailed information

```
ranch:info().
```

The `ranch:procs/2` function will return all acceptor or listener processes for a given listener.

Get all acceptor processes

```
ranch:procs(tcp_echo, acceptors).
```

Get all connection processes

```
ranch:procs(tcp_echo, connections).
```

Chapter 3

Transports

A transport defines the interface to interact with a socket.

Transports can be used for connecting, listening and accepting connections, but also for receiving and sending data. Both passive and active mode are supported, although all sockets are initialized as passive.

3.1 TCP transport

The TCP transport is a thin wrapper around `gen_tcp`.

3.2 SSL transport

The SSL transport is a thin wrapper around `ssl`.

Ranch depends on `ssl` by default so any necessary dependencies will start when Ranch is started. It is possible to remove the dependency when the SSL transport will not be used. Refer to your release build tool's documentation for more information.

When embedding Ranch listeners that have an SSL transport, your application must depend on the `ssl` application for proper behavior.

3.3 Sending and receiving data

This section assumes that `Transport` is a valid transport handler (like `ranch_tcp` or `ranch_ssl`) and `Socket` is a connected socket obtained through the listener.

You can send data to a socket by calling the `Transport:send/2` function. The data can be given as `iodata()`, which is defined as `binary() | iolist()`. All the following calls will work:

Sending data to the socket

```
Transport:send(Socket, <<"Ranch is cool!">>).
Transport:send(Socket, "Ranch is cool!").
Transport:send(Socket, ["Ranch", ["is", "cool!"]]).
Transport:send(Socket, ["Ranch", [<<"is">>, "cool!"]]).
```

You can receive data either in passive or in active mode. Passive mode means that you will perform a blocking `Transport:recv/3` call, while active mode means that you will receive the data as a message.

By default, all data will be received as binary. It is possible to receive data as strings, although this is not recommended as binaries are a more efficient construct, especially for binary protocols.

Receiving data using passive mode requires a single function call. The first argument is the socket, and the third argument is a timeout duration before the call returns with `{error, timeout}`.

The second argument is the amount of data in bytes that we want to receive. The function will wait for data until it has received exactly this amount. If you are not expecting a precise size, you can specify 0 which will make this call return as soon as data was read, regardless of its size.

Receiving data from the socket in passive mode

```
{ok, Data} = Transport:recv(Socket, 0, 5000).
```

Active mode requires you to inform the socket that you want to receive data as a message and to write the code to actually receive it.

There are two kinds of active modes: `{active, once}` and `{active, true}`. The first will send a single message before going back to passive mode; the second will send messages indefinitely. We recommend not using the `{active, true}` mode as it could quickly flood your process mailbox. It's better to keep the data in the socket and read it only when required.

Three different messages can be received:

- `{OK, Socket, Data}`
- `{Closed, Socket}`
- `{Error, Socket, Reason}`

The value of `OK`, `Closed` and `Error` can be different depending on the transport being used. To be able to properly match on them you must first call the `Transport:messages/0` function.

Retrieving the transport's active message identifiers

```
{OK, Closed, Error} = Transport:messages().
```

To start receiving messages you will need to call the `Transport:setopts/2` function, and do so every time you want to receive data.

Receiving messages from the socket in active mode

```
{OK, Closed, Error} = Transport:messages(),
Transport:setopts(Socket, [{active, once}]),
receive
    {OK, Socket, Data} ->
        io:format("data received: ~p~n", [Data]);
    {Closed, Socket} ->
        io:format("socket got closed!~n");
    {Error, Socket, Reason} ->
        io:format("error happened: ~p~n", [Reason])
end.
```

You can easily integrate active sockets with existing Erlang code as all you really need is just a few more clauses when receiving messages.

3.4 Sending files

As in the previous section it is assumed `Transport` is a valid transport handler and `Socket` is a connected socket obtained through the listener.

To send a whole file, with name `Filename`, over a socket:

Sending a file by filename

```
{ok, SentBytes} = Transport:sendfile(Socket, Filename).
```

Or part of a file, with `Offset` greater than or equal to 0, `Bytes` number of bytes and chunks of size `ChunkSize`:

Sending part of a file by filename in chunks

```
Opts = [{chunk_size, ChunkSize}],
{ok, SentBytes} = Transport:sendfile(Socket, Filename, Offset, Bytes, Opts).
```

To improve efficiency when sending multiple parts of the same file it is also possible to use a file descriptor opened in raw mode:

Sending a file opened in raw mode

```
{ok, RawFile} = file:open(Filename, [raw, read, binary]),
{ok, SentBytes} = Transport:sendfile(Socket, RawFile, Offset, Bytes, Opts).
```

3.5 Upgrading a TCP socket to SSL

A connected TCP socket can be upgraded to a SSL socket via the function `ranch_ssl:handshake/3`. The socket **must** be in `{active, false}` mode before telling the client that the server is ready to upgrade in order to avoid race conditions.

Performing a TLS handshake on a TCP socket

```
{ok, NewSocket} = ranch_ssl:handshake(Socket, SslOpts, 5000).
```

3.6 Writing a transport handler

A transport handler is a module implementing the `ranch_transport` behavior. It defines a certain number of callbacks that must be written in order to allow transparent usage of the transport handler.

The behavior doesn't define the socket options available when opening a socket. These do not need to be common to all transports as it's easy enough to write different initialization functions for the different transports that will be used. With one exception though. The `setopts/2` function **must** implement the `{active, once}` and the `{active, true}` options.

If the transport handler doesn't have a native implementation of `sendfile/5` a fallback is available, `ranch_transport:sendfil`
The extra first argument is the transport's module. See `ranch_ssl` for an example.

Chapter 4

Protocols

A protocol handler starts a connection process and defines the protocol logic executed in this process.

4.1 Writing a protocol handler

All protocol handlers must implement the `ranch_protocol` behavior which defines a single callback, `start_link/4`. This callback is responsible for spawning a new process for handling the connection. It receives four arguments: the name of the listener, the socket, the transport handler being used and the protocol options defined in the call to `ranch:start_listener/5`. This callback must return `{ok, Pid}`, with `Pid` the pid of the new process.

The newly started process can then freely initialize itself. However, it must call `ranch:handshake/1,2` before doing any socket operation. This will ensure the connection process is the owner of the socket. It expects the listener's name as argument.

Perform the socket handshake

```
{ok, Socket} = ranch:handshake(Ref).
```

If your protocol code requires specific socket options, you should set them while initializing your connection process, after calling `ranch:handshake/1,2`. You can use `Transport:setopts/2` for that purpose.

Following is the complete protocol code for the example found in `examples/tcp_echo/`.

Protocol module that echoes everything it receives

```
-module(echo_protocol).
-behaviour(ranch_protocol).

-export([start_link/4]).
-export([init/3]).

start_link(Ref, _Socket, Transport, Opts) ->
    Pid = spawn_link(?MODULE, init, [Ref, Transport, Opts]),
    {ok, Pid}.

init(Ref, Transport, _Opts = []) ->
    {ok, Socket} = ranch:handshake(Ref),
    loop(Socket, Transport).

loop(Socket, Transport) ->
    case Transport:recv(Socket, 0, 5000) of
        {ok, Data} ->
            Transport:send(Socket, Data),
            loop(Socket, Transport);
        _ ->
```

```
            ok = Transport:close(Socket)
    end.
```

4.2 Using gen_statem

Special processes like the ones that use the `gen_statem` or `gen_server` behaviours have the particularity of having their `start_link` call not return until the `init` function returns. This is problematic, because you won't be able to call `ranch:handshak` from the `init` callback as this would cause a deadlock to happen.

Use the `gen_statem:enter_loop/4` function. It allows you to start your process normally (although it must be started with `proc_lib` like all special processes), then perform any needed operations before falling back into the normal `gen_statem` execution loop.

Use a gen_statem for protocol handling

```
-module(my_protocol).
-behaviour(gen_statem).
-behaviour(ranch_protocol).

-export([start_link/4]).
-export([init/1]).
%% Exports of other gen_statem callbacks here.

start_link(Ref, _Socket, Transport, Opts) ->
    {ok, proc_lib:spawn_link(?MODULE, init, [{Ref, Transport, Opts}])}.

init({Ref, Transport, _Opts = []}) ->
    %% Perform any required state initialization here.
    {ok, Socket} = ranch:handshake(Ref),
    ok = Transport:setopts(Socket, [{active, once}]),
    gen_statem:enter_loop(?MODULE, [], state_name, {state_data, Socket, Transport}).

%% Other gen_statem callbacks here.
```

Check the `tcp_reverse` example for a complete example.

Chapter 5

Embedded mode

Embedded mode allows you to insert Ranch listeners directly in your supervision tree. This allows for greater fault tolerance control by permitting the shutdown of a listener due to the failure of another part of the application and vice versa.

5.1 Embedding

To embed Ranch in your application you can simply add the child specs to your supervision tree. This can all be done in the `init/1` function of one of your application supervisors.

Ranch requires at the minimum two kinds of child specs for embedding. First, you need to add `ranch_sup` to your supervision tree, only once, regardless of the number of listeners you will use. Then you need to add the child specs for each listener.

Ranch has a convenience function for getting the listeners child specs called `ranch:child_spec/5`, that works like `ranch:start_` except that it doesn't start anything, it only returns child specs.

As for `ranch_sup`, the child spec is simple enough to not require a convenience function.

The following example adds both `ranch_sup` and one listener to another application's supervision tree.

Embed Ranch directly in your supervision tree

```
init([]) ->
    RanchSupSpec = {ranch_sup, {ranch_sup, start_link, []},
        permanent, 5000, supervisor, [ranch_sup]},
    ListenerSpec = ranch:child_spec(echo, 100,
        ranch_tcp, [{port, 5555}],
        echo_protocol, []
    ),
    {ok, {{one_for_one, 10, 10}, [RanchSupSpec, ListenerSpec]}}.
```

Remember, you can add as many listener child specs as needed, but only one `ranch_sup` spec!

It is recommended that your architecture makes sure that all listeners are restarted if `ranch_sup` fails. See the Ranch internals chapter for more details on how Ranch does it.

Part II

How to

Chapter 6

Writing parsers

There are three kinds of protocols:

- Text protocols
- Schema-less binary protocols
- Schema-based binary protocols

This chapter introduces the first two kinds. It will not cover more advanced topics such as continuations or parser generators.

This chapter isn't specifically about Ranch, we assume here that you know how to read data from the socket. The data you read and the data that hasn't been parsed is saved in a buffer. Every time you read from the socket, the data read is appended to the buffer. What happens next depends on the kind of protocol. We will only cover the first two.

6.1 Parsing text

Text protocols are generally line based. This means that we can't do anything with them until we receive the full line.

A simple way to get a full line is to use `binary:split/2,3`.

Using binary:split/2 to get a line of input

```
case binary:split(Buffer, <<"\n">>) of
    [_] ->
        get_more_data(Buffer);
    [Line, Rest] ->
        handle_line(Line, Rest)
end.
```

In the above example, we can have two results. Either there was a line break in the buffer and we get it split into two parts, the line and the rest of the buffer; or there was no line break in the buffer and we need to get more data from the socket.

Next, we need to parse the line. The simplest way is to again split, here on space. The difference is that we want to split on all spaces character, as we want to tokenize the whole string.

Using binary:split/3 to split text

```
case binary:split(Line, <<" ">>, [global]) of
    [<<"HELLO">>] ->
        be_polite();
    [<<"AUTH">>, User, Password] ->
        authenticate_user(User, Password);
    [<<"QUIT">>, Reason] ->
        quit(Reason)
    %% ...
end.
```

Pretty simple, right? Match on the command name, get the rest of the tokens in variables and call the respective functions.

After doing this, you will want to check if there is another line in the buffer, and handle it immediately if any. Otherwise wait for more data.

6.2 Parsing binary

Binary protocols can be more varied, although most of them are pretty similar. The first four bytes of a frame tend to be the size of the frame, which is followed by a certain number of bytes for the type of frame and then various parameters.

Sometimes the size of the frame includes the first four bytes, sometimes not. Other times this size is encoded over two bytes. And even other times little-endian is used instead of big-endian.

The general idea stays the same though.

Using binary pattern matching to split frames

```
<< Size:32, _/bits >> = Buffer,
case Buffer of
    << Frame:Size/binary, Rest/bits >> ->
        handle_frame(Frame, Rest);
    _ ->
        get_more_data(Buffer)
end.
```

You will then need to parse this frame using binary pattern matching, and handle it. Then you will want to check if there is another frame fully received in the buffer, and handle it immediately if any. Otherwise wait for more data.

Chapter 7

SSL client authentication

7.1 Purpose

SSL client authentication is a mechanism allowing applications to identify certificates. This allows your application to make sure that the client is an authorized certificate, but makes no claim about whether the user can be trusted. This can be combined with a password based authentication to attain greater security.

The server only needs to retain the certificate serial number and the certificate issuer to authenticate the certificate. Together, they can be used to uniquely identify a certicate.

As Ranch allows the same protocol code to be used for both SSL and non-SSL transports, you need to make sure you are in an SSL context before attempting to perform an SSL client authentication. This can be done by checking the return value of `Transport:name/0`.

7.2 Obtaining client certificates

You can obtain client certificates from various sources. You can generate them yourself, or you can use a service like CAcert.org which allows you to generate client and server certificates for free.

Following are the steps you need to take to create a CAcert.org account, generate a certificate and install it in your favorite browser.

- Open http://cacert.org in your favorite browser
- Root Certificate link: install both certificates
- Join (Register an account)
- Verify your account (check your email inbox!)
- Log in
- Client Certificates: New
- Follow instructions to create the certificate
- Install the certificate in your browser

You can optionally save the certificate for later use, for example to extract the `IssuerID` information as will be detailed later on.

7.3 Transport configuration

The SSL transport does not request a client certificate by default. You need to specify the `{verify, verify_peer}` option when starting the listener to enable this behavior.

Configure a listener for SSL authentication

```
{ok, _} = ranch:start_listener(my_ssl,
    ranch_ssl, [
        {port, SSLPort},
        {certfile, PathToCertfile},
        {cacertfile, PathToCACertfile},
        {verify, verify_peer}
    ],
    my_protocol, []
).
```

In this example we set the required `port` and `certfile`, but also the `cacertfile` containing the CACert.org root certificate, and the option to request the client certificate.

If you enable the `{verify, verify_peer}` option and the client does not have a client certificate configured for your domain, then no certificate will be sent. This allows you to use SSL for more than just authenticated clients.

7.4 Authentication

To authenticate users, you must first save the certificate information required. If you have your users' certificate files, you can simply load the certificate and retrieve the information directly.

Retrieve the issuer ID from a certificate

```
certfile_to_issuer_id(Filename) ->
    {ok, Data} = file:read_file(Filename),
    [{'Certificate', Cert, not_encrypted}] = public_key:pem_decode(Data),
    {ok, IssuerID} = public_key:pkix_issuer_id(Cert, self),
    IssuerID.
```

The `IssuerID` variable contains both the certificate serial number and the certificate issuer stored in a tuple, so this value alone can be used to uniquely identify the user certificate. You can save this value in a database, a configuration file or any other place where an Erlang term can be stored and retrieved.

To retrieve the `IssuerID` from a running connection, you need to first retrieve the client certificate and then extract this information from it. Ranch does not provide a function to retrieve the client certificate. Instead you can use the `ssl:peercert/1` function. Once you have the certificate, you can again use the `public_key:pkix_issuer_id/2` to extract the `IssuerID` value.

The following function returns the `IssuerID` or `false` if no client certificate was found. This snippet is intended to be used from your protocol code.

Retrieve the issuer ID from the certificate for the current connection

```
socket_to_issuer_id(Socket) ->
    case ssl:peercert(Socket) of
        {error, no_peercert} ->
            false;
        {ok, Cert} ->
            {ok, IssuerID} = public_key:pkix_issuer_id(Cert, self),
            IssuerID
    end.
```

You then only need to match the `IssuerID` value to authenticate the user.

Part III

Advanced

Chapter 8

Internals

This chapter may not apply to embedded Ranch as embedding allows you to use an architecture specific to your application, which may or may not be compatible with the description of the Ranch application.

Note that for everything related to efficiency and performance, you should perform the benchmarks yourself to get the numbers that matter to you. Generic benchmarks found on the web may or may not be of use to you, you can never know until you benchmark your own system.

8.1 Architecture

Ranch is an OTP application.

Like all OTP applications, Ranch has a top supervisor. It is responsible for supervising the `ranch_server` process and all the listeners that will be started.

The `ranch_server` gen_server is a central process keeping track of the listeners and their acceptors. It does so through the use of a public ets table called `ranch_server`. The table is owned by the top supervisor to improve fault tolerance. This way if the `ranch_server` gen_server fails, it doesn't lose any information and the restarted process can continue as if nothing happened.

Ranch uses a custom supervisor for managing connections. This supervisor keeps track of the number of connections and handles connection limits directly. While it is heavily optimized to perform the task of creating connection processes for accepted connections, it is still following the OTP principles and the usual `sys` and `supervisor` calls will work on it as expected.

Listeners are grouped into the `ranch_listener_sup` supervisor and consist of three kinds of processes: the listener gen_server, the acceptor processes and the connection processes, both grouped under their own supervisor. All of these processes are registered to the `ranch_server` gen_server with varying amount of information.

All socket operations, including listening for connections, go through transport handlers. Accepted connections are given to the protocol handler. Transport handlers are simple callback modules for performing operations on sockets. Protocol handlers start a new process, which receives socket ownership, with no requirements on how the code should be written inside that new process.

8.2 Number of acceptors

The second argument to `ranch:start_listener/5` is the number of processes that will be accepting connections. Care should be taken when choosing this number.

First of all, it should not be confused with the maximum number of connections. Acceptor processes are only used for accepting and have nothing else in common with connection processes. Therefore there is nothing to be gained from setting this number too high, in fact it can slow everything else down.

Second, this number should be high enough to allow Ranch to accept connections concurrently. But the number of cores available doesn't seem to be the only factor for choosing this number, as we can observe faster accepts if we have more acceptors than cores. It might be entirely dependent on the protocol, however.

Our observations suggest that using 100 acceptors on modern hardware is a good solution, as it's big enough to always have acceptors ready and it's low enough that it doesn't have a negative impact on the system's performances.

8.3 Platform-specific TCP features

Some socket options are platform-specific and not supported by `inet`. They can be of interest because they generally are related to optimizations provided by the underlying OS. They can still be enabled thanks to the `raw` option, for which we will see an example.

One of these features is `TCP_DEFER_ACCEPT` on Linux. It is a simplified accept mechanism which will wait for application data to come in before handing out the connection to the Erlang process.

This is especially useful if you expect many connections to be mostly idle, perhaps part of a connection pool. They can be handled by the kernel directly until they send any real data, instead of allocating resources to idle connections.

To enable this mechanism, the following option can be used.

Using raw transport options

```
{raw, 6, 9, << 30:32/native >>}
```

It means go on layer 6, turn on option 9 with the given integer parameter.

Part IV

Additional information

Appendix A

Upcoming changes in Ranch 2.0

The following changes will be done in Ranch 2.0. In most cases an alternative is already available in the most recent Ranch version.

- The function `ranch:start_listener/6` has been deprecated in favor of `ranch:start_listener/5`. The number of acceptors was removed and will be taken from the transport options.
- The function `ranch:child_spec/6` has also been deprecated, in favor of `ranch:child_spec/5`.
- The function `ranch:accept_ack/1` has been deprecated in favor of `ranch:handshake/1,2`.
- The function `ranch:info/1,2` will return a map containing each listener's information rather than a list of key/values. The `num_acceptors` key will be removed.
- The socket will no longer be passed to the protocol when starting it. It will be available as a return value from `ranch:handshake/1,` only.
- Starting from Ranch 2.0 it will no longer be allowed to pass Ranch options along with socket options as a proplist. The only forms allowed will be the `ranch:opts()` map or socket options as-is. The `ranch:opts()` map must be used in case socket options also use a map.
- The `socket` option will be removed. A more viable solution is to define a custom transport module that returns a fresh socket when `Transport:listen/1` is called.

Appendix B

Changes since Ranch 1.8

Ranch 1.8 is the last Ranch 1.x release series. Only compatibility fixes are to be released moving forward. Support for Ranch 1.8 will be discontinued when Cowboy 3.0 is released.

The following patch releases have been issued:

B.1 Ranch 1.8.1

- Fix `ranch:recv_proxy_header/2` for the upcoming Erlang/OTP 28.
- Erlang.mk has been updated.

Appendix C

Migrating from Ranch 1.7 to 1.8

Ranch 1.8 is a compatibility update for Erlang/OTP 24.

Ranch 1.8 is compatible with Erlang/OTP 21.0 onward. Support for Erlang/OTP 19 and 20 has been removed.

C.1 Bugs fixed

- An issue with the PROXY protocol was fixed in Ranch 1.7.1. The wrong CRC32 algorithm was used and would cause checksum verification to fail when used. The configuration value when building PROXY headers has been changed to `crc32c` to reflect the correct algorithm.

Appendix D

Migrating from Ranch 1.6 to 1.7

Ranch 1.7 adds built-in support for the PROXY protocol.

The PROXY protocol is a simple and efficient way for proxies to transmit information about the client.

While a third-party library already existed, it was not entirely compatible with the Ranch interface, in particular when socket active mode was involved. This new implementation fixes that and supports the full protocol with as little overhead as possible compared to normal operations: just one extra function call.

Ranch 1.7 is compatible with Erlang/OTP 19.0 onward. Support for Erlang/OTP 18 has been removed.

D.1 Features added

- Full support for the PROXY protocol was added.

D.2 New functions

- Add the function `ranch:recv_proxy_header/2` to receive the PROXY protocol header and parse it. It must be called before `ranch:handshake/1,2`.
- Add the functions `ranch_proxy_header:parse/1` and `ranch_proxy_header:header/1,2` to parse and build a PROXY protocol header, respectively.

D.3 Bugs fixed

- Fix a race condition when the listener is restarted after `ranch_listener_sup` crashes. This resulted in the original options being used even if the options were updated at runtime.
- Make the acceptors exit instead of crash when the listening socket has been closed to prevent unnecessary logs.
- Fix an issue where listener information would not get cleaned up when an embedded listener was stopped. This was fixed in Ranch 1.6.2.

Appendix E

Migrating from Ranch 1.5 to 1.6

Ranch 1.6 added the ability to suspend and resume listeners. It also deprecates a number of features and add interfaces that will be used in Ranch 2.0.

Ranch 1.6 is compatible with Erlang/OTP 18.0 onward. Support for older releases has been removed.

E.1 Features added

- Listeners can now be suspended/resumed without stopping existing connection processes. This effectively closes the listening socket and stops the acceptor processes.
- Transport options can now be updated for suspended listeners.
- The `Socket` argument given when the protocol starts has been deprecated. In Ranch 2.0 the socket will be obtainable only by calling `ranch:handshake/1,2`.
- Ranch-specific transport options and socket options are now better separated. When passing Ranch-specific transport options, Ranch now expects to receive a map, in which case socket options are passed in the `socket_opts` value. When there are only socket options they can be passed to Ranch directly as a convenience.
- Any future transport option will only be added to the map type. This includes transport options added in this release.
- The transport option `ack_timeout` was renamed to `handshake_timeout` in the map type.
- The `cacerts` socket option is now silenced in error logs just like the `certs` and `key` options.
- The manual has been heavily updated and now features one manual page per function and module, complete with a per-function changelog, examples and more.

E.2 Experimental features added

- It is now possible to configure the restart intensity for `ranch_sup` using the OTP application environment. This feature will remain undocumented unless there is popular demand for it.
- Add the transport option `logger` that allows configuring which logger module will be used. The logger module must follow the interface of the new `logger` module in Erlang/OTP 21, or be set to `error_logger` to keep the old behavior.

E.3 Changed behaviors

- Transport modules must now implement `Transport:handshake/2,3` which deprecates and will replace `Transport:accept` in Ranch 2.0. It returns a new socket and can therefore be used for implementing TLS upgrade mechanisms.

E.4 New functions

- The functions `ranch:suspend_listener/1` and `ranch:resume_listener/1` were added. In addition the function `ranch:get_state/1` can be used to obtain the running state of a listener.
- A function `ranch:wait_for_connections/3` was added.
- A function `ranch:handshake/1,2` was added to replace the function `ranch:accept_ack/1`.

E.5 Bugs fixed

- The specs for the function `Transport:sendfile/2,4,5` have been corrected. The type used for the filename was too restricting.

Appendix F

Migrating from Ranch 1.x

The changelog for Ranch releases before 1.6 can be found in this section.

F.1 1.5.0

- Add transport functions getopts/2, getstat/1 and getstat/2
- Fix ranch:info/0 and ranch:procs/2 in embedded mode
- Prevent ranch_conns_sup from stopping on unexpected messages

F.2 1.4.0

- Add new transport option num_acceptor
- Deprecate ranch:start_listener/6 in favor of start_listener/5
- Deprecate ranch:child_spec/6 in favor of child_spec/5

F.3 1.3.0

The version numbers 1.3.1 and 1.3.2 were later made to fix small mistakes made during the 1.3.0 release process. They do not include code changes.

- Tested with OTP R16B+ on Linux, FreeBSD, OSX and Windows
- Add ssl to the list of dependencies
- Add ranch:info/0 and ranch:procs/2 to retrieve Ranch state information
- Allow configuring a listener with only SNI, without a default certificate
- Blacklist transport options instead of whitelist
  - Unknown options are now allowed, but will result in a Dialyzer warning
- Add many transport options typespecs and documentation
- Don't silently drop the accept rate when running out of fds

- Prevent a race condition when stopping listeners
- Improve reporting for common errors, for example eaddrinuse
- Fix double removal of connections bug
  - The number of active connections should now be exact
- Fix stuck acceptor bug when controlling_socket returned errors
- Numerous documentation and examples improvements

F.4 1.2.1

- Fix bug preventing node shutdown when SSL is used with OTP 17.1+
- Tune restart intensity in all supervisors

F.5 1.2.0

- Allow the supervised process and the process owning the socket to be different
- Add many transport options (please refer to the documentation)
- Add function ranch:get_addr/1 to retrieve both IP and port of listener
- Don't pass Ranch-specific options down to transports
  - Should make Dialyzer happy in user projects
  - New types ranch:opt(), ranch_tcp:opt(), ranch_ssl:ssl_opt() and ranch_ssl:opt()
- Fix crash when filtering unknown options out
- Print a warning for each option filtered out
- Handle Transport:controlling_socket/2 errors and close the socket
- Handle Protocol:start_link/4 crashes to avoid killing all active connections
- Use Asciidoc for documentation
- Test Ranch across 14 Erlang versions on CircleCI
- Improve and document test suites with recent ct_helper improvements
- Fix a number of intermittent test issues

F.6 1.1.0

- Add Transport:secure/0
- Add SSL partial_chain option
- Stop reporting errors on {error, closed} in accept_ack

F.7 1.0.0

- Initial release