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 . . . 4
2.5 Listening on a random port . . . 5
2.6 Listening on privileged ports . . . 5
2.7 Listening on a UNIX Domain socket . . . 5
2.8 Performing additional setup steps on a listening socket . . . 5
2.9 Accepting connections on an existing socket . . . 6
2.10 Limiting the number of concurrent connections . . . 6
2.11 Customizing the number of acceptor processes . . . 7
2.12 Customizing the number of connection supervisors . . . 7
2.13 Setting connection count alarms . . . 8
2.14 When running out of file descriptors . . . 8
2.15 Using a supervisor for connection processes . . . 8
2.16 Upgrading . . . 9
2.17 Changing transport options . . . 9
2.18 Obtaining information about listeners . . . 9

3 Transports **11**
3.1 TCP transport . . . 11
3.2 SSL transport . . . 11
3.3 Sending and receiving data . . . 11
3.4 Sending files . . . 12
3.5 Upgrading a TCP socket to SSL . . . 13
3.6 Writing a transport handler . . . 13

4 Protocols **14**
4.1 Writing a protocol handler . . . 14
4.2 Using gen_statem and gen_server . . . 15
4.2.1 gen_statem . . . 15
4.2.2 gen_server . . . 15

5 Embedded mode **16**
5.1 Embedding . . . 16

II How to **17**

6 Writing parsers **18**
6.1 Parsing text . . . 18
6.2 Parsing binary . . . 19

7 SSL client authentication **20**
7.1 Purpose . . . 20
7.2 Obtaining client certificates . . . 20
7.3 Transport configuration . . . 21
7.4 Authentication . . . 21

8 Connection draining **22**

III Advanced **24**

9 Internals **25**
9.1 Architecture . . . 25
9.2 Number of acceptors . . . 25
9.3 Platform-specific TCP features . . . 26

IV Additional information **27**

A Migrating from Ranch 2.3 to 2.4 **28**
A.1 Features added . . . 28
A.2 Bugs fixed . . . 28

B Migrating from Ranch 2.2 to Ranch 2.3 **29**
B.1 Features added . . . 29
B.2 Bugs fixed . . . 29

C Migrating from Ranch 2.1 to Ranch 2.2 **30**
C.1 Features added . . . 30
C.2 Changed behaviors . . . 30
C.3 Bugs fixed . . . 30

D Migrating from Ranch 2.0 to Ranch 2.1 **31**
D.1 Features added . . . 31
D.2 Experimental features added . . . 31
D.3 Changed behaviors . . . 31
D.4 Added functions . . . 32

E Migrating from Ranch 1.7+ to Ranch 2.0 **33**
E.1 Features added . . . 33
E.2 Experimental features added . . . 33
E.3 Features removed . . . 34
E.4 Changed behaviors . . . 34
E.5 Added functions . . . 34
E.6 Changed functions . . . 34
E.7 Removed functions . . . 34
E.8 Bugs fixed . . . 34
E.9 Other changes . . . 35

F Migrating from Ranch 1.6 to 1.7 **36**
F.1 Features added . . . 36
F.2 New functions . . . 36
F.3 Bugs fixed . . . 36

G Migrating from Ranch 1.5 to 1.6 **37**
G.1 Features added . . . 37
G.2 Experimental features added . . . 37
G.3 Changed behaviors . . . 37
G.4 New functions . . . 38
G.5 Bugs fixed . . . 38

H Migrating from Ranch 1.x **39**

H.1 1.5.0 . . . 39

H.2 1.4.0 . . . 39

H.3 1.3.0 . . . 39

H.4 1.2.1 . . . 40

H.5 1.2.0 . . . 40

H.6 1.1.0 . . . 40

H.7 1.0.0 . . . 40

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, macOS and Windows.

Ranch is developed for Erlang/OTP 27+.

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.
- A transport handler and its associated options.
- A protocol handler and its associated options.

Transport options are documented in ranch(3).

Ranch includes both TCP and SSL transport handlers, respectively ranch_tcp(3) and ranch_ssl(3).

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 10 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, #{socket_opts => [{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}` and `{reuseaddr, true}` set. These values can't be overridden when starting the listener, but they can be overridden using `Transport:setopts/2` in the protocol.

It will also set `{backlog, 1024}`, `{nodelay, true}`, `{send_timeout, 30000}` and `{send_timeout_close, true}`, which can be overridden at listener startup. These four options are not applied for DTLS.

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, #{socket_opts => [{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 Listening on a UNIX Domain socket

On UNIX systems, it is also possible to use Ranch to listen on a UNIX domain socket by specifying `{local, SocketFile}` for the `ip` socket option. In this case, the port must be set to 0 or omitted. Before listening, and again from `ranch:stop_listener/1` Ranch deletes a socket or FIFO at that path. A regular file, directory, symlink, or device is left in place and must not already exist.

Starting a listener for TCP connections on a UNIX Domain socket

```
{ok, _} = ranch:start_listener(tcp_echo,
    ranch_tcp, #{socket_opts => [
        {ip, {local, "/tmp/ranch_echo.sock"}},
        {port, 0}
    ]}, echo_protocol, []
).
```

2.8 Performing additional setup steps on a listening socket

If it is necessary to perform additional setup steps on the listening socket, it is possible to specify a function with the transport option `post_listen_callback`. This function will be called after the listening socket has been created but before accepting connections on it, with the socket as the single argument.

The function must return either the atom `ok`, after which the listener will start accepting connections on the socket, or a tuple `{error, Reason}` which will cause the listener to fail starting with `Reason`.

Setting permissions on a UNIX Domain socket file

```
PostListenCb = fun (Sock) ->
    case ranch_tcp:sockname(Sock) of
        {ok, {local, SockFile}} ->
            file:change_mode(SockFile, 8#777);
    {ok, _} ->
        ok;
    Error = {error, _} ->
            Error
    end
end,

{ok, _} = ranch:start_listener(tcp_echo,
    ranch_tcp, #{
        socket_opts => [
            {ip, {local, "/tmp/ranch_echo.sock"}},
            {port, 0}],
        post_listen_callback => PostListenCb},
    echo_protocol, []
).
```

2.9 Accepting connections on an existing socket

If you want to accept connections on an existing socket, you can write a custom ranch_transport(3) implementation that fetches or otherwise acquires a listening socket in the `listen/1` callback and returns it in the form of `{ok, ListenSocket}`.

The custom `listen/1` function must ensure that the listener process (usually the process calling it) is also made the controlling process of the socket it returns. Failing to do so will result in stop/start and suspend/resume not working properly for that listener.

2.10 Limiting the number of concurrent connections

The `max_connections` transport option allows you to limit the number of concurrent connections per connection supervisor (see below). 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, #{socket_opts => [{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, #{socket_opts => [{port, 5555}], max_connections => infinity},
    echo_protocol, []
).
```

The maximum number of connections is a soft limit per connection supervisor. In practice, it can reach `max_connections` plus the number of acceptors assigned to that supervisor.

When the maximum number of connections is reached, Ranch will stop accepting connections. The kernel queues incoming connections until the queue is full. The size of this queue is determined by the `backlog` option and defaults to 1024. Once the queue is full, the kernel drops or rejects further connections. 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).
```

When the socket belongs to a child of the supervised process, call `ranch:remove_connection/2` with that parent pid instead.

Removing a supervised connection from the count

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

As seen in the chapter covering protocols, this reference is received as the first argument of the protocol's `start_link/3` 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.

2.11 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, #{socket_opts => [{port, 5555}], num_acceptors => 42},
    echo_protocol, []
).
```

2.12 Customizing the number of connection supervisors

By default Ranch will use one connection supervisor for each acceptor process. An acceptor has only one connection supervisor, but a connection supervisor can have more than one acceptor. Their task is to supervise the connection processes started by an acceptor. The number of connection supervisors can be tweaked.

Note that the association between the individual acceptors and connection supervisors is fixed, meaning that acceptors will always use the same connection supervisor to start connection processes.

Specifying a custom number of connection supervisors

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

2.13 Setting connection count alarms

The `alarms` transport option allows you to configure alarms which will be triggered when the number of connections tracked by one connection supervisor reaches or exceeds the defined threshold.

The `alarms` transport option takes a map with alarm names as keys and alarm options as values.

Any term is allowed as an alarm name.

Alarm options include the alarm type and a threshold that, when reached, triggers the given callback. A cooldown prevents the alarm from being triggered too often.

Log warnings when the number of connections is at or above 100

```
Alarms = #{
    my_alarm => #{
        type => num_connections,
        threshold => 100,
        callback => fun(Ref, AlarmName, ConnSup, ConnPids) ->
            logger:warning("Alarm ~0tp: listener ~0tp "
                    "supervisor ~0tp has ~b connections",
                [AlarmName, Ref, ConnSup, length(ConnPids)])
        end
    }
},
{ok, _} = ranch:start_listener(tcp_echo,
    ranch_tcp, #{alarms => Alarms, socket_opts => [{port, 5555}]},
    echo_protocol, []
).
```

In the example code, an alarm named `my_alarm` is defined, which will call the given function when the number of connections tracked by the connection supervisor reaches or exceeds 100. When the number of connections is still (or again) at or above 100 after the default cooldown period of 5 seconds, the alarm will trigger again.

2.14 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.

```
=WARNING REPORT==== 19-Oct-2026::16:31:45.128603 ===
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.

`emfile` and `enfile` are the only accept errors Ranch logs. `ranch:trace_acceptors/1,2` samples every reason returned by `Transport:accept/2`.

2.15 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}`.

The count is stored under `SupPid`. The protocol process removes that connection from the count by calling `ranch:remove_connec` `SupPid)`.

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.

2.16 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).
```

2.17 Changing transport options

Ranch allows you to change the transport options of a listener with the ranch:set_transport_options(3) function, for example to change the number of acceptors or to make it listen on a different port. See that manual for when each option takes effect.

Changing the transport options

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

The number of acceptors and the socket options, including the port, apply only after the listener is suspended and resumed.

Suspending, changing transport options and resuming

```
ok = ranch:suspend_listener(tcp_echo),
ok = ranch:set_transport_options(tcp_echo, NewOpts),
ok = ranch:resume_listener(tcp_echo).
```

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).
```

2.18 Obtaining information about listeners

Ranch provides 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 connection processes for a given listener.

Get all acceptor processes

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

Get all connection processes

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

`ranch:trace_acceptors/1,2` samples the `{error, Reason}` tuples returned by `Transport:accept/2`. Call it from a shell to see why a listener fails to accept connections.

Sample accept errors

```
ranch:trace_acceptors(tcp_echo).
```

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 three kinds of active modes: `{active, once}`, `{active, N}` and `{active, true}`. The first will send a single message before going back to passive mode; the second will send `N` messages followed by a `Passive` message when switching back to passive mode; the third 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.

Four different messages can be received:

- Incoming data: `{OK, Socket, Data}`
- Socket closed: `{Closed, Socket}`
- Socket error: `{Error, Socket, Reason}`
- Switch to passive mode: `{Passive, Socket}`

The value of `OK`, `Closed`, `Error` and `Passive` 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, Passive} = 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, Passive} = 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.



Important
The new socket received from `ranch_ssl:handshake/3` must be used via the `ranch_ssl` transport.

Performing a TLS handshake on a TCP socket

```
{ok, SslSocket} = ranch_ssl:handshake(TcpSocket, 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:sendfile(3). The extra first argument is the transport's module. See `ranch_ssl` for an example.

It is highly recommended for a custom transport handler to implement the optional `format_error/1` callback, in order to provide a human-readable diagnostic string. Implementing this callback will become mandatory in Ranch 3.0.

The `peercert/1` and `negotiated_protocol/1` callbacks are optional, as they only apply to TLS.

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/3`. This callback is responsible for spawning a new process for handling the connection. It receives three arguments: the name of the listener, 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. When `connection_type` is `supervisor` it may instead return `{ok, SupPid, ConnPid}`. See ranch_protocol(3).

The newly started process can then freely initialize itself. It must call ranch:handshake(3) before doing any socket operation. This makes the connection process the owner of the socket. The argument is the listener name.

When a PROXY protocol header is expected, call ranch:recv_proxy_header(3) before `ranch:handshake/1,2`.

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

start_link(Ref, 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 and gen_server

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.

This problem can be addressed in several ways.

4.2.1 gen_statem

- Use state enter calls and place the `ranch:handshake/1,2` call in the enter clause of the initial state. Check the `tcp_reverse` example for a complete example.
- Use a `next_event` action in the return from `init/1` and place the `ranch:handshake/1,2` call in the clause handling the event in the initial state.
- Use the `gen_statem:enter_loop/4` function and start your process with `proc_lib:spawn_link/3` or `proc_lib:start` See below for an example.

Using gen_statem:enter_loop/4 to start a protocol

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

-export([start_link/3]).
-export([init/1]).
%% Exports of other gen_statem callbacks here.

start_link(Ref, 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.
```

4.2.2 gen_server

- Use `{continue, Continue}` in the return from `init/1` and place the `ranch:handshake/1,2` call in a corresponding `handle_continue/2` clause.
- Use the `gen_server:enter_loop/3` function and start your process with `proc_lib:spawn_link/3` or `proc_lib:start`

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.

However, just as for non-embedded listeners that were started via `ranch:start_listener/5`, it is required that the `ranch` application is running before you can start embedded listeners. Furthermore, this also means that embedded listeners will restart when `ranch_sup` fails.



Warning
By using embedded mode, it is possible to start a listener with the same name as an already existing listener. This will corrupt the information Ranch keeps for that listener, so you should take care to ensure unique listener names yourself. A good way to achieve this is by combining the embedded listener's name with `?MODULE`, or the name of the application it is used in.

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 has a convenience function for getting the listeners child specs called ranch:child_spec(3), that works like `ranch:start_liste` except that it doesn't start anything, it only returns child specs.

The following example adds one listener to another application's supervision tree.

Embed Ranch directly in your supervision tree

```
init([]) ->
    ListenerSpec = ranch:child_spec({?MODULE, echo},
        ranch_tcp, #{socket_opts => [{port, 5555}]},
        echo_protocol, []
    ),
    {ok, {#{}, [ListenerSpec]}}.
```

Embedded listeners cannot be stopped via `ranch:stop_listener/1`. Instead, they are to be stopped as part of the shutdown of your application's supervision tree.

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

```
case Buffer of
    << Size:32, _/bits >> when byte_size(Buffer) >= Size ->
        << Frame:Size/binary, Rest/bits >> = Buffer,
        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 certificate.

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 https://www.cacert.org in your favorite browser
- Root Certificate: install the Class 1 and Class 3 certificates
- Join (Register an account)
- Verify your account (check your email inbox!)
- Log in
- Client Certificates: New
- Create the private key and the certificate request locally, then submit the request
- 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. `{verify, verify_peer}` requests one and rejects a certificate that does not chain to `cacertfile`.

Configure a listener for SSL authentication

```
{ok, _} = ranch:start_listener(my_ssl,
    ranch_ssl, #{socket_opts => [
        {port, SSLPort},
        {certfile, PathToCertfile},
        {cacertfile, PathToCACertfile},
        {verify, verify_peer},
        {fail_if_no_peer_cert, false}
    ]},
    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.

`{fail_if_no_peer_cert, false}`, as in the example, lets the handshake complete when the client sends no certificate. `Transport:peercert/1` returns `{error, no_peercert}`. 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. Call `Transport:peercert/1`. Once you have the certificate, you can again use `public_key:pkix_issuer_id/1` 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(Transport, Socket) ->
    case Transport: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.

Chapter 8

Connection draining

Stopping a Ranch listener via `ranch:stop_listener/1` closes the listening socket and then stops its connection processes. They receive a shutdown signal and are killed if still running after the `shutdown` transport option, which defaults to 5000 ms. `brutal_kill` kills them immediately. However, you may want to stop a listener in a graceful fashion, i.e. by not accepting any new connections, but allowing the existing connection processes to exit by themselves instead of being killed.

For this purpose, you should first suspend the listener you wish to stop gracefully, and then wait for its connection count to drop to zero by calling ranch:wait_for_connections(3).

Draining a single listener

```
ok = ranch:suspend_listener(Ref),
ok = ranch:wait_for_connections(Ref, '==', 0),
ok = ranch:stop_listener(Ref).
```

If you want to drain more than just one listener, it may be important to first suspend them all before beginning to wait for their connection counts to reach zero. Otherwise, the not yet suspended listeners will still be accepting connections while you wait for the suspended ones to be drained.

Draining multiple listeners

```
lists:foreach(
    fun (Ref) ->
        ok = ranch:suspend_listener(Ref)
    end,
    Refs
),
lists:foreach(
    fun (Ref) ->
        ok = ranch:wait_for_connections(Ref, '==', 0),
        ok = ranch:stop_listener(Ref)
    end,
    Refs
).
```

If you have long-running connection processes hosted by the listener you want to stop gracefully, draining may take a long time, possibly forever. If you just want to give the connection processes a chance to finish, but are not willing to wait for infinity, the waiting part could be handled in a separate process.

Draining a listener with a timeout

```
ok = ranch:suspend_listener(Ref),
{DrainPid, DrainRef} = spawn_monitor(
    fun () ->
        ok = ranch:wait_for_connections(Ref, '==', 0)
    end
```

```
),
receive
    {'DOWN', DrainRef, process, DrainPid, _} ->
        ok
after DrainTimeout ->
    exit(DrainPid, kill),
    ok
end,
ok = ranch:stop_listener(Ref).
```

To drain listeners automatically as part of your application shutdown routine, use the `prep_stop/1` function of your application module.

Draining listeners automatically on application shutdown

```
-module(my_app).

-behavior(application).

-export([start/2]).
-export([prep_stop/1]).
-export([stop/1]).

start(_StartType, _StartArgs) ->
    {ok, _} = ranch:start_listener(my_listener, ranch_tcp, #{}, my_protocol, []),
    my_app_sup:start_link().

prep_stop(State) ->
    ok = ranch:suspend_listener(my_listener),
    ok = ranch:wait_for_connections(my_listener, '==', 0),
    ok = ranch:stop_listener(my_listener),
    State.

stop(_State) ->
    ok.
```

Part III

Advanced

Chapter 9

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.

9.1 Architecture

Ranch is an OTP application.

Like all OTP applications, Ranch has a top supervisor, `ranch_sup`. It supervises the `ranch_server` process. Listener supervisors are added as children of `ranch_sup` when `ranch:start_listener/5` is called.

The `ranch_server` gen_server keeps track of listener information through a public ETS table, also named `ranch_server`. `ranch_app` creates that table before starting `ranch_sup`, so the application master owns it. If `ranch_server` fails, the table remains and the restarted process continues as if nothing happened.

Listener, acceptor and connection supervisor processes set a `proc_lib` label that includes the listener ref.

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.

Each listener is a `ranch_listener_sup` supervisor. It supervises `ranch_conns_sup_sup`, which supervises the connection supervisors, and `ranch_acceptors_sup`, which supervises the acceptor processes. There is no per-listener gen_server. The listener supervisor and the connection supervisors record their pids in the `ranch_server` table.

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.

9.2 Number of acceptors

The `num_acceptors` transport option is the number of processes that will be accepting connections. It defaults to 10. 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.

9.3 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

Migrating from Ranch 2.3 to 2.4

Ranch 2.4 adds PROXY protocol controls and acceptor diagnostics, and fixes connection accounting, local socket cleanup, and PROXY header parsing.

Ranch 2.4 is tested with Erlang/OTP 27.0 onward.

A.1 Features added

- The `trusted_proxies` transport option limits which immediate peers may send a PROXY protocol header. Each entry is `{inet:ip4_address(), 0..32}` or `{inet:ip6_address(), 0..128}`. When the option is absent, any immediate peer may send a header. An empty list rejects every peer.
- `ranch:remove_connection/2` drops the given pid when the process is under a `supervisor`.
- Optional transport callbacks `peercert/1` and `negotiated_protocol/1` were added. `ranch_ssl` implements them.
- PROXY protocol version 2 parses and builds the SSL TLV subtypes `group`, `sig_scheme`, and `client_cert`. `client_cert` carries the client certificate in ASN.1 DER form.
- `ranch:trace_acceptors/1,2` samples the `{error, Reason}` values from `Transport:accept/2`.

A.2 Bugs fixed

- Potential incompatibility: Before listening, and again from `ranch:stop_listener/1`, Ranch deletes a `{local, Path}` file only when it is a socket or FIFO. A regular file, directory, symlink, or device is left in place and must not already exist.
- Potential incompatibility: `ranch:child_spec/5` raises `{badarg, {bad_option, Key}}` when a transport option is invalid.
- Potential incompatibility: A PROXY protocol version 1 line longer than 107 bytes is rejected before it is decoded. This includes a TCP6 line whose addresses would otherwise parse.
- PROXY protocol version 2 reading and parsing was strenghtened.
- `ranch:set_transport_options/2` left the stored `max_connections` limit unchanged. `ranch:get_max_connectio` and a restarted connection supervisor now see the new value.
- A release upgrade now fails when an acceptor does not stop or restart using the Ranch-provided appup files.

Appendix B

Migrating from Ranch 2.2 to Ranch 2.3

Ranch 2.3 is a maintenance release containing a small number of fixes and improvements.

Ranch 2.3 is tested with Erlang/OTP 27.0 onward.

B.1 Features added

- Support for Erlang/OTP versions below 27.0 was dropped. Ranch will now only support Erlang/OTP versions that receive security patches.
- Ranch processes now have labels.

B.2 Bugs fixed

- Unnecessary crash logs were sometimes produced during PROXY header parsing. This has been addressed.
- Acceptors now handle `emfile` and `enfile` errors the same way, instead of just handling `emfile`. `enfile` is the system-wide equivalent of `emfile`.

Appendix C

Migrating from Ranch 2.1 to Ranch 2.2

Ranch 2.2 is a maintenance release containing a small number of fixes and improvements.

Ranch 2.2 is tested with Erlang/OTP 24.0 onward, although it should be compatible with Erlang/OTP 22.0 and above.

C.1 Features added

- `DTLS` is now supported.
- The `certs_keys` option from `ssl` is now accepted and documented.
- The `UNIQUE_ID` PROXY protocol header extension is now supported.
- The `ranch_transport` behavior has a new callback `format_error/1` which allows finer grained formatting of Ranch's error messages. The callback is currently optional and will become required in Ranch 3.0.

C.2 Changed behaviors

- Ranch will now obfuscate certificates, keys and passwords in the error message produced on listen error.
- The exception reason when a connection process has failed its handshake has been changed to `{shutdown, {Reason, PeerInfo}}` where `PeerInfo` contains the peer name when available.
- The `{packet, raw}` socket option is no longer set explicitly, as this is already the default value.

C.3 Bugs fixed

- `ranch:stop_listener/1` will now return an error instead of throwing an exception when the listener does not exist.
- Fix `ranch:recv_proxy_header/2` for the upcoming Erlang/OTP 28.
- Ensure that a user crash while stopping a listener does not prevent a subsequent attempt to stop it.
- Alarm option `threshold` was misspelled in Ranch 2.1. This has been corrected. The wrong spelling `treshold` is still accepted and will be removed in Ranch 3.0.

Appendix D

Migrating from Ranch 2.0 to Ranch 2.1

Ranch 2.1 adds counters and alarms.

The Prometheus collector was updated to include accepted/terminated connections metrics.

Ranch 2.1 is compatible with Erlang/OTP 22.0 onward. Support for Erlang/OTP 21 has been removed.

D.1 Features added

- Metrics are now provided by `ranch:info/0,1`. Currently includes accepted/terminated connection counts per connection supervisor.
- Alarms can now be configured. The only alarm currently available is `num_connections`. When the number of connections goes over a configurable threshold Ranch will call the given callback. This can be used to programmatically shut down idle connections to make up space for new connections, for example.
- A `post_listen_callback` option has been added. It receives sockets immediately after the `Transport:listen/1` call. It can be used for some additional initialization of the socket, such as setting file permissions on Unix domain sockets.
- It is now possible to use TLS-PSK authentication without having to specify a default certificate for TLS < 1.3.

D.2 Experimental features added

- The `inet_backend` option is now properly handled and tested for TCP listeners. This allows using the experimental `socket` backend. The `socket` backend is now tested with Ranch. Note that there are known issues and Windows support is not currently implemented.

D.3 Changed behaviors

- Ranch will now remove unsupported SSL/TLS options where applicable. A warning will be logged when this happens. Options are only removed when they are not compatible with the selected TLS version and leaving them would prevent the listener from starting.

```
The following options are removed when using TLS
1.1, 1.2 or 1.3: `beast_mitigation` and `padding_check`.
```

```
The following options are removed when using TLS
1.3 exclusively: `client_renegotiation`,
`next_protocols_advertised`, `psk_identity`,
`reuse_session`, `reuse_sessions`,
`secure_renegotiate` and `user_lookup_fun`.
```

D.4 Added functions

- The function `ranch_proxy_header:to_connection_info/1` converts PROXY protocol information to the same format as `ssl:connection_information/1`. Because there is little overlap only the `protocol`, `selected_cipher_suit` and `sni_hostname` will be available, however.

Appendix E

Migrating from Ranch 1.7+ to Ranch 2.0

Ranch 2.0 adds support for multiple connection supervisors.

Ranch 1.x had a bottleneck because it used only a single connection supervisor. This was more evident when many connections were dropped at once as the supervisor couldn't keep up and failed to accept new connections while cleaning up the old ones. Ranch 2.0 behaves much better in this scenario by default. Multiple connection supervisors also helps with concurrently accepting new connections.

Ranch 2.0 also adds experimental support for opening more than one listening socket on a single port.

Starting with Ranch 2.0 we are also providing a Prometheus collector as a separate project as well as a Grafana dashboard.

Ranch 2.0 is compatible with Erlang/OTP 21.0 onward. Support for Erlang/OTP 19 and 20 has been removed.

E.1 Features added

- Ranch now comes with a `ranch.appup` file necessary for performing release upgrades. A test suite has been added to confirm release upgrades work from one tag to the next. Numerous fixes were made that will also improve error recovery. Release upgrades will only be supported from Ranch 2.0 onward.
- The `num_conns_sups` option has been added. It allows configuring the number of connection supervisors. It now defaults to `num_accceptors`. The old behavior can be obtained by setting this value to 1.
- The `logger` option is no longer experimental. It now defaults to `logger` instead of `error_logger`.
- UNIX domain sockets are now supported.
- The active N socket option is now supported. It requires Erlang/OTP 21.3 or above for TLS, however.
- Embedded listeners are now failing in a predictable manner when `ranch_server` goes down. It is no longer necessary to embed `ranch_sup` and the recommendation is now to just start Ranch normally when using embedded listeners.
- Two steps handshake is now supported. This allows obtaining TLS extensions and updating options before resuming the handshake. The handshake can also be canceled.

E.2 Experimental features added

- The experimental `num_listen_sockets` option has been added. It allows opening more than one listening socket per listener. It can only be used alongside the Linux `SO_REUSEPORT` socket option or equivalent. It allows working around a bottleneck in the kernel and maximizes resource usage, leading to increased rates for accepting new connections.

E.3 Features removed

- The `socket` option was removed. A more viable solution is to define a custom transport module that returns a fresh socket when `Transport:listen/1` is called.

E.4 Changed behaviors

- The callback function `Transport:listen/1` and its implementations in `ranch_tcp` and `ranch_ssl` have changed to accept a map of transport options instead of only socket options.
- The callback function `Transport:messages/0` return value now includes the tag used for passive messages.
- The `Socket` argument was removed from `Protocol:start_link/3`. The socket must now be obtained by calling `ranch:handshake/1,2`.

E.5 Added functions

- The functions `ranch:handshake_continue/1,2` and `ranch:handshake_cancel/1` can be used to perform a two steps handshake. These functions may not be supported by all transports.

E.6 Changed functions

- The `NumAcceptors` argument was removed from `ranch:start_listener/5` and `ranch:child_spec/5` and moved to the transport options.
- Ranch options can no longer be passed along with socket options as a proplist. The only forms allowed are now the `ranch:opts()` map or only socket options as-is. Individual transport options are now validated as well. The `ranch:opts()` map must be used when socket options also use a map. This applies to the `ranch:start_listener/5`, `ranch:child_spec/5` and `ranch:set_transport_options/2` functions.
- The function `ranch:info/1,2` now returns a map containing each listener's information rather than a list of key/values. The key `num_acceptors` was removed as it can be found in the transport options.
- The function `ranch:set_transport_options/2` no longer requires the listener to be suspended. Which options apply immediately, on suspend/resume or on restart has been documented. Some work has also been done to make these option changes more predictable.

E.7 Removed functions

- The function `ranch:accept_ack/1` has been removed in favor of `ranch:handshake/1,2`.

E.8 Bugs fixed

- Calling `ranch:remove_connection/1` will now resume a sleeping acceptor process when applicable.
- Repeatedly calling `ranch:remove_connection/1` from a connection process would crash the respective connection supervisor. This has now been fixed.
- When a connection process was failing to start, the socket was not closed and this lead to leaking sockets. This is now corrected.

E.9 Other changes

- Connection draining has now been documented in the guide following user feedback and discussions.
- Ranch is now tested against Concuerror, a model checking tool for debugging, testing and verifying concurrent Erlang programs. Two tests have been added in this release and more will follow in the future.
- Ranch is now tested against `stampede`, a chaos monkey style testing tool. Currently includes three scenarios: normal TCP and TLS listeners and embedded TCP listener. This new test suite helped uncover a misplaced `monitor/2` call added during the development of Ranch 2.0 (we were using a similar tool, `havoc`, at the time of finding that issue).
- The supervisor for acceptors and the parent supervisor for connection supervisors now have an adaptive restart intensity limit set to `1 + ceil(math:log2(NumChildren))` to allow room for errors when they have many children.
- Ranch now uses stricter compiler options. Missing function specs were added to internal modules.
- Ranch now calls `ssl:handshake/1,2,3` instead of `ssl:ssl_accept/1,2`.
- The `ranch_ssl:ssl_opt()` type has been updated to conform with Erlang/OTP 23.0.

Appendix F

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.

F.1 Features added

- Full support for the PROXY protocol was added.

F.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.

F.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 G

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.

G.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.

G.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.

G.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.

G.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`.

G.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 H

Migrating from Ranch 1.x

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

H.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

H.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

H.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

H.4 1.2.1

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

H.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

H.6 1.1.0

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

H.7 1.0.0

- Initial release