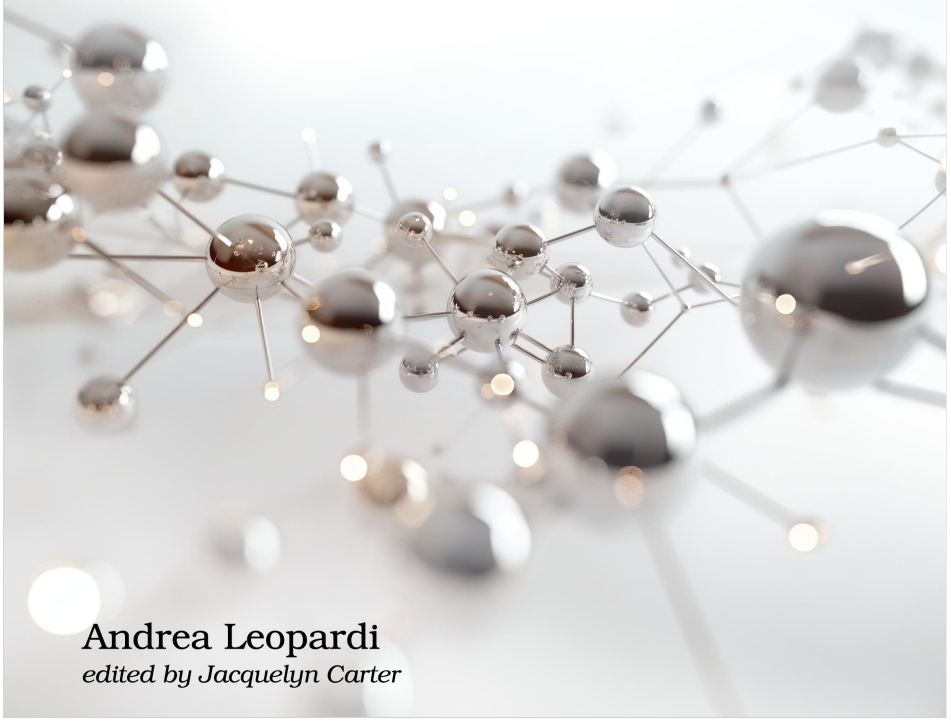
The Pragmatic Programmers

Your Elixir Source

Network Programming in Elixir and Erlang

Write High-Performance, Scalable, and Reliable Apps with TCP and UDP



Andrea Leopardi
edited by Jacquelyn Carter

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Queuing Requests

Right now, our client is not really able to deal with two callers checking out the socket simultaneously. We're about to address this issue.

First and foremost, let's verify that this is an actual issue. We can use Task to spawn two processes that send a command to Redis through the client at the same time. Fire up IEx and type this in:

```
iex> {:ok, client} = RedisClient.start_link(host: 'localhost', port: 6379)
iex> for _ <- 1..2 do
...>   Task.async(fn -> RedisClient.command(client, ["PING"]) end)
...> end
```

You will likely see some error, like a FunctionClauseError or a CaseClauseError. Our client code is full of race conditions when dealing with multiple callers!

The problem is that we are allowing multiple callers to check out the socket at the same time. When this happens, we run into cases where multiple callers call :gen_tcp.send/2 to send their commands, which results in multiple responses being sent by Redis. However, remember that only one process can be calling :gen_tcp.recv/3 at any given time on a socket. This means that you'll likely see one of the two tasks get both responses from the :gen_tcp.recv/3 call, while the other would see a {:error, :ealready} error.

These issues make sense though. If we want to check out the TCP socket to callers, we must only hand the socket to one caller at a time. We could return an error if a caller tries to check out the socket when the socket is already checked out, but that would make for a pretty awkward API. The most common approach to deal with this is *queuing requests* instead.

Implementing a Simple Queue (That Doesn't Handle Reconnections)

Let's implement a simple and naive caller queue. For now, we'll completely ignore *error states*, that is, what happens when the connection gets interrupted for some reason. We'll dig deeper into that in the next chapter, where we'll come up with strategies to handle reconnections and pending requests. So, let's go ahead and focus on the queueing for now. We'll start by changing the state to include a queue of callers. Erlang's standard library provides a queue module[5] that we can use.

```
-defstruct [:host, :port, :socket, :caller_monitor]
+defstruct [:host, :port, :socket, :caller_monitor, queue: :queue.new()]
```

5. https://www.erlang.org/doc/man/queue.html

The elements of this queue will be the from tuples that we get in the handle_call(:checkout, from, state) callback. Those are a good choice because we can then use GenServer.reply/2 to reply to callers once the socket is available, instead of returning {:reply, {:ok, socket}, state} from handle_call/3.

Now, we need to queue callers when handling checkout calls if the socket is already checked out. We'll do two things. First, we'll "enqueue" the new caller. Then, we'll call a helper function which takes the first caller out of the queue and hands it the socket. We're using a helper function because we'll be able to re-use it in other parts of the client, as you'll see in a moment. Let's look at the code to handle the :checkout call.

redis_client/lib/redis_client_queue.ex

```
def handle_call(:checkout, _from, %__MODULE__{socket: nil} = state) do
  {:reply, {:error, :not_connected}, state}
end

def handle_call(:checkout, from, %__MODULE__{} = state) do
  # First, queue the new caller.
  state = update_in(state.queue, &:queue.in(from, &1))

  # Then, check out the next queued caller.
  state = checkout_if_waiting(state)
  {:noreply, state}
end
```

The helper function we mentioned is checkout_if_waiting/1, which we define as:

redis_client/lib/redis_client_queue.ex

```
Line 1  # If there is already a caller which checked out the socket,
-       # leave the state alone.
-       defp checkout_if_waiting(%__MODULE__{caller_monitor: ref} = state)
-            when is_reference(ref) do
5         state
-       end
-
-       # If we can hand the socket to a waiting caller, we do that. If there
-       # are no callers waiting, we just return the state.
10      defp checkout_if_waiting(%__MODULE__{} = state) do
-         case :queue.out(state.queue) do
-           {:empty, _empty_queue} ->
-             state
-
15         {{:value, {pid, _ref} = from}, new_queue} ->
-            ref = Process.monitor(pid)
-            :inet.setopts(state.socket, active: false)
-            GenServer.reply(from, {:ok, state.socket})
-            %__MODULE__{state | queue: new_queue, caller_monitor: ref}
20        end
-       end
```

Most of this code is made of pieces of code that we already saw in this section. In this helper function, we use :queue.out/1 to get the next element out of the callers queue, on line 11. The return value of :queue.out/2 is either {:empty, queue} if the queue is empty, or {{:value, item}, new_queue}, with item being the next item in the queue. On line 18, you can see how we call GenServer.reply/2 to hand the socket over to the waiting from caller.

Now, we can modify how we handle :checkin calls as well. The only difference is that we need to check if there are waiting callers in the queue once the current caller checks in the socket. Luckily, we can reuse the checkout_if_waiting/1 helper we just saw. Let's look at the code for the handle_call/3 clause in question.

redis_client/lib/redis_client_queue.ex

```
def handle_call(:checkin, _from, %__MODULE__{} = state) do
  Process.demonitor(state.caller_monitor, [:flush])
  state = %__MODULE__{state | caller_monitor: nil}
  state = checkout_if_waiting(state)
  {:reply, :ok, state}
end
```

Our client can now handle a queue of callers waiting for their turn to check out the socket. However, this means that if the connection is interrupted for some reason, then we likely want to also notify the waiting callers about it. We can define a helper for this that we'll call when handling :tcp_closed and :tcp_error messages. Here's the code for this helper:

```
# If the socket is closed, we reply to all the waiting callers,
# because we don't know if we'll ever reconnect. Callers can always
# retry if needed.
defp flush_queue(state) do
  Enum.each(:queue.to_list(state.queue), fn from ->
    GenServer.reply(from, {:error, :disconnected})
  end)

  %__MODULE__{state | queue: :queue.new()}
end
```

The idea is to reply to all the waiting callers with {:error, :disconnected} if the connection goes down, so that those callers can decide what to do in case that happens. For example, some callers might decide to retry the command they were trying to send, while others might decide to "give up".

We've got a working Redis client that can queue requests from callers and handle Redis commands. Awesome job! Let's end this section with a short overview of what we built and some thoughts on this "check-out/check-in" kind of client.

Pros and Cons of Checkout Clients

Let's remember how we got here. Our goal was to build a basic Redis client. To do that, we went with an approach where our client is essentially a BEAM process that controls a TCP client socket. Other processes can use the client to interact with Redis. The way they do that is to *check out* the TCP socket from the client, and interact with it directly for as long as needed before checking it back in.

After the first pass at this approach, we added *request queuing*. Our final client is able to also serve multiple processes interacting with Redis by queuing their requests and handing over the socket to one client at a time. We also baked quite a resilient approach to *error handling* in the Redis client. The client is able to deal with disconnections by notifying waiting callers and attempting to reconnect after a short backoff period. Our client is also able to deal with callers crashing by monitoring the current caller, and going through a reconnection cycle if the current caller goes down.

The main advantage of this approach to TCP clients on the BEAM is that we move the TCP socket around instead of *moving the data* around. On the BEAM, sending a message from a process to another generally means copying the data being sent from the memory of the sending process to the memory of the receiving process. The only case when this is not true is for binaries larger than 64 bytes, which live in a common memory space. This is a desirable property in a virtual machine like ours, because it allows processes to be completely isolated and allows the VM to garbage-collect processes individually. However, a TCP client like the one we built is mostly sending messages to and from caller processes, so the memory overhead of always copying those messages can be impactful. Moving the socket around, instead, means that callers can read off of and write to the OS socket buffer directly, without copying data within the BEAM.

How the BEAM Stores Binaries in Memory

Most Erlang terms live in the memory of each process using them. One exception is binaries. Small binaries below 64 bytes in size are stored in the process heap (*heap binaries*), but binaries larger than that are stored in a shared memory area. Those are called *refc binaries* (*reference-counted binaries*) because they are referenced by processes using them, and the BEAM counts the references to each of them to know when to garbage-collect them. Another representation of binaries on the BEAM is *sub binaries*, which are objects that can reference only *part* of a refc or heap binary. For example, you can use `:binary.referenced_byte_size/1`[a] to get the size of the referenced binary if the given binary is a sub binary.

Knowing this information about binaries is important when designing the process architecture of your application and planning how your processes will exchange messages. For example, if you expect most messages with a TCP server to be large binaries, then you could avoid passing the socket around, since the binaries will likely be refc binaries anyway. That means that sending them across processes will only send a reference to shared memory. You can read in-detail information about binaries and their performance in the Erlang efficiency guide.[b]

a. https://www.erlang.org/doc/man/binary.html#referenced_byte_size-1
b. https://www.erlang.org/doc/efficiency_guide/binaryhandling.html

This approach has one drawback when it comes to data though: our client is not taking advantage of *TCP multiplexing*. Multiplexing is a property of TCP which essentially means that a TCP socket can receive *and* send data simultaneously. In our client, we only ever receive *or* send data, since we always send a command with `:gen_tcp.send/2`, and only then start receiving a response with `:gen_tcp.recv/3`. This is fine with the way our client operates, since we only ever give the socket to one caller at a time. However, we could take advantage of TCP multiplexing if we allowed multiple clients to send their commands concurrently. To do that, we'd have to receive all data in a single process and dispatch the right responses to the right callers, though. This is a viable alternative approach, and we'll explore it in the next chapter.

A second drawback, related to *resilience*, is the potential size of the callers queue. As a general rule, whenever your system has a queue, you'll likely want to enforce an *upper bound* on the size of that queue. Not doing that can potentially result in memory leaks. In our client, we could choose to limit the size of the callers queue, and either drop older callers or stop accepting new callers until more caller requests are served.

While it took us a good while to go through the code for the Redis client, we managed to get a resilient client working in under two-hundred lines of code (including comments!). However, in a real-world system you'd be unlikely to use a client like this directly. This is due to one main reason: performance. Whenever your system is interacting with an external system (such as Redis), you'll likely want to connect to that system using a *pool* of connections. This is the same idea as the TCP acceptor pool that we explored in Increasing Scalability with Multiple Acceptors, on page ?. We'll explore pooling TCP client sockets in the next chapter.