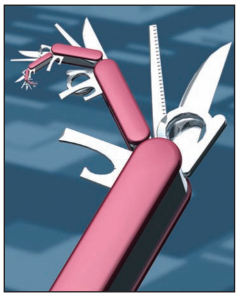

Process Bottlenecks within Erlang Web Applications

Steve Vinoski • *Verivue*

Writing server-side Web applications and Web servers in Erlang is pretty straightforward. The Web protocol is, of course, HTTP over TCP, for which Erlang provides comprehensive built-in support. Web server applications are naturally concurrent owing to the need to simultaneously handle requests from multiple clients, and Erlang provides applications with remarkably painless concurrency support. Such applications also need access to back-end data stores and networked services, and Erlang supplies them with multiple database choices and flexible support for integrating with non-Erlang service applications. In addition, various Erlang Web servers, libraries, and frameworks provide applications with template languages and support for common representation formats such as HTML, XML, and JavaScript Object Notation (JSON).

As with any programming language and runtime system, Erlang has constructs and approaches that, if applied incorrectly, can negatively affect application performance and scalability. Developers typically find poorly performing applications frustrating and discouraging, especially if they're new to the language. Being aware of a language's performance traps and how to avoid them is key for newcomers investigating different languages in the hopes of finding advantages not only in performance, but also for ease of development and maintenance, flexibility, and time to market.

When compared to the concurrency constructs used in most other languages, Erlang processes are simple yet powerful. Even so, new Erlang developers often misuse them and fail to understand how and where underlying frameworks and libraries use processes. In the context of Erlang Web servers and Web applications, failing to properly manage processes and communication between them can result in systems that perform and scale poorly.

Anatomy of an Erlang Web Server

Because structural awareness of a typical Erlang Web server is important for understanding how to properly use Erlang processes, let's begin by writing parts of a simple Web server. If you're familiar with regular socket programming, you'll likely recognize my approach's organization, even if you're not familiar with Erlang.

First, you need a simple function to listen for incoming connections (see Figure 1). This code listens for connections on port 8000 and invokes the `loop/1` function, passing the listen socket. (Erlang functions are identified using the form "name/arity," where arity refers to the number of function arguments.)

The `loop/1` function in Figure 2 is also straightforward. This function creates a new process that executes the `acceptor/2` function to accept incoming client connections, and then waits for the newly spawned process to notify it when a new client has arrived. It passes the listen socket and its own process identifier (retrieved via the `self()` function) so that the acceptor process knows whom to notify when it accepts a new connection. After receiving the `accepted` message, `loop/1` calls itself recursively to create a new acceptor process.

Figure 3 shows the acceptor function. It accepts an incoming client connection, represented by a socket `Sock`. Then, it sends the `loop/1` process the `accepted` message so that it knows to create a new acceptor. It then proceeds to handle the request. I'll cover the `handle_request/1` function implementation later.

Although I haven't shown anything yet that's specific to Web servers — the listening and accepting loop could serve any TCP-based protocol — what's important is that this design uses a process-per-connection approach. This approach, typical of Erlang Web servers, means that the system handles every new connection within a separate Erlang process. Taking such an approach with operating system threads would be a disaster owing to their high cost, but Erlang processes are extremely lightweight and are both inexpensive and fast to create. An Erlang runtime system can easily support hundreds of thousands of concurrent processes within a single operating system process. This makes it not only easy but practical for our simple Web server to spawn a new process every time it gets a new connection, let the new process handle that connection, and then let the process die when the connection closes.

```
start() ->
  ListenOptions = [binary, {reuseaddr, true}],
  {ok, LS} = gen_tcp:listen(8000, ListenOptions),
  loop(LS).
```

Figure 1. A simple listening function. This function sets up a listen socket for connections on port 8000. Once the socket is set up, this function passes it to a loop that accepts incoming client connections on that port.

```
loop(LS) ->
  spawn(?MODULE, acceptor, [LS, self()]),
  receive
    accepted -> loop(LS)
  end.
```

Figure 2. The `loop/1` function. This tail-recursive function spawns a new process to accept incoming connections, and then waits for that process to send it a message indicating it's accepted a connection. Once it receives that message, it calls itself recursively to create a new acceptor.

```
acceptor(LS, Parent) ->
  {ok, Sock} = gen_tcp:accept(LS),
  Parent ! accepted,
  handle_request(Sock).
```

Figure 3. The acceptor function. This function accepts a new connection, tells its parent process (running the `loop/1` function) that it's accepted a new connection, and then proceeds to handle the request.

Managing State

If the server-side application running under the simple Web server uses databases or back-end services to help fulfill requests, you need a way to hold onto the database connection or the network connection to the back-end service. Because Erlang doesn't support global or per-module variables, you can store the state required to hold such connections in a process.

The Erlang open source distribution (available from www.erlang.org) includes a set of libraries and frameworks called the Open Telecom Platform (OTP), so named because it was initially created to support the development of telecommunications systems such as telephone switches. OTP — which, despite its name, turned out to be useful for all types of applications — provides state-holding extensible process frameworks called *behaviors*. Applications extend OTP behaviors by implementing specific callback functions that help handle messages sent to the behavior process. For example, the `gen_server` behavior supplies basic support, letting applications write server processes, and the `gen_fsm` behavior lets applications easily implement finite-state machines.

Fundamentally, OTP behaviors operate as processes executing tail-recursive functions that receive and act on messages. These functions also store state in a variable that each function passes to its next recursive invocation. A behavior's workflow generally proceeds in a manner similar to that shown in Figure 4.

The `behavior_loop` function takes a single argument: the state carried by the process running the loop. The function first receives a message and passes that message along with the current state to the `app_callback` function, which the application using the behavior supplies. The callback function processes the message and returns a 2-tuple consisting of a loop directive and new state. The loop directive tells the loop what to do next: if it's the atom `stop`, the loop stops; otherwise, the loop calls itself recursively, passing the new state. The recursion keeps the process from ending until it's explicitly stopped.

Actual OTP behaviors such as `gen_server` and `gen_fsm` are more sophisticated than this because they support multiple application callback functions and multiple loop directives, provide debugging facilities, and help handle live runtime code updates. Even so, a behavior is essentially a process maintaining state via tail-recursive function calls and receiving messages that inform the application's callbacks of what actions to perform next, acting on the state as they carry out the desired action.

```
behavior_loop(State) ->
  receive
    Message ->
      {Next, NewState} = app_callback(Message, State),
      case Next of
        stop ->
          ok;
        _Else ->
          behavior_loop(NewState)
      end
  end.
```

Figure 4. A behavior's conceptual workflow. A behavior's core is a tail-recursive function loop that passes state to each recursion. The loop's body receives messages and invokes application callback functions to act on them. The application callbacks return new state for the next recursion as well as directives to tell the loop what to do next. Actual OTP behaviors are noticeably more sophisticated than this simple example.

```
handle_request(Sock) ->
    Request = read_request(Sock),
    Response = gen_server:call(proxy, Request),
    return_response(Sock, Response).
```

Figure 5. The application's request handling function. After reading the request from the socket, the function calls a `gen_server` process that in turn invokes a back-end service to fulfill the request. The `gen_server` keeps the network connection to the back-end service in its state. This function uses the `gen_server`'s response to reply to the HTTP client.

Request Serialization

A developer following normal Erlang best practices would naturally employ a `gen_server` or `gen_fsm` to hold a connection to a database or back-end server in the loop state. Messages or calls into the `gen_server` or `gen_fsm` would direct it to send requests over its connection to store or retrieve information to or from the database or service at the other end. Unfortunately, this approach won't work well in the context of this simple Web server.

Recall that this simple Web server creates a new Erlang process for each new connection. Let's say the Web server application uses a `gen_server` to hold a connection to a back-end networked service. The application handles incoming HTTP requests by sending requests to the service to retrieve information that it then uses to create HTTP responses. As I detailed earlier, the Web server initially handles each HTTP request in a separate acceptor process that calls `handle_request/1`. You might implement a `handle_request/1` function like the one in Figure 5.

The `handle_request/1` function reads the request from the socket and passes it to a call to the back-end service proxy, registered in the local Erlang process registry under the name `proxy`, which is the `gen_server` that holds the service connection. The actual back-end service returns response data to the `gen_server`, which in turn processes that data to return a response to `handle_request/1`. The `handle_request/1` function uses the response to reply to the HTTP client. The invocation of the `gen_server:call` function is special in that it exchanges messages with the `gen_server` process, rather than being just a simple function call. It first sends a message into the state-holding `gen_server` process to ask it to carry out a request to the back-end service. Then, the `gen_server:call` waits in a `receive` for a default of five seconds in the caller's process for the `gen_server` to send back a response message. In other words, the `gen_server:call` here involves two processes: the caller's process and the called `proxy` process (the `gen_server`).

Now, consider what happens if 1,000 Web clients connect at about the same time, each issuing a `GET` request. The Web server creates a new acceptor process for each Web client connection, 1,000 in all. Each acceptor calls the `handle_request/1` function, which reads the incoming request from the socket, forms a request message, and invokes the `gen_server:call` to send the request message to the `proxy` process. In Erlang, each process has a queue in which it receives messages sent by other processes; the 1,000 acceptor processes thus put a total of 1,000 messages into the `proxy` process message queue. The `proxy` process then proceeds to drain its queue by receiving request messages one by one in its behavior loop. It invokes an application callback for each one to handle the request. The callback calls over to the actual back-end service, gets a reply, and then sends the reply back to the calling acceptor process for return to the Web client. Figure 6 illustrates the `acceptor` processes and the `proxy` process.

The problem is that this design provides the opposite of a shared-nothing architecture: the `proxy` process message queue effectively serializes all HTTP requests. The first few of the 1,000 requests might get processed relatively quickly, but the deeper in the `proxy` message queue a request is, the longer it takes to be handled. As the number of concurrent client requests increases, not only does latency rise owing to increased `proxy` message queue length, but queues can grow without bound because Erlang process queues offer no back pressure to sender processes. A fast sender not only overruns a receiver, it can even place enough messages in the receiver's queue to use up all available memory and crash the entire Erlang runtime system.

Better Approaches

This solution's significant shortcomings become obvious if you measure throughput and latency of the application under varying client loads. They're also obvious if you're not an Erlang beginner and you understand the internals of your Erlang Web server and OTP behavior internals. But, if you're a newcomer to Erlang and you measure the system, there's a good chance your inexperience with the language might lead you to incorrectly conclude that the problems are inherent in Erlang and are thus unavoidable.

A variety of better approaches exist, but not surprisingly, they all involve reducing or eliminating contention for resources. Which one is best depends on your particular system's characteristics. For example, creating a pool of `proxy` instances would let the acceptor processes randomly load-balance requests across the pool. Another option that would work with either a single `proxy` instance or a pool is to maintain a local cache of `proxy` results, thereby allowing many requests to completely bypass access to the back-end service or database. Cached results could be stored in an Erlang ets (Erlang term storage) table, which is an in-memory store that multiple Erlang processes can read and write concurrently.

The ideal approach is to fully process each HTTP request and response completely in the Web server process that accepted the connection. If you can't obtain this ideal for your application, it might be possible to decompose the logic in a `proxy` instance into smaller shared state-holding processes, each responsible for part of the work. This approach spreads the load of all the Web server processes among these finer-grained resource managers, breaking up the original `proxy`'s single coarse-grained point of contention. This not only reduces overall contention but also lets each Web server process use only the exact shared resources required to fulfill a particular request. It also makes the thread of control clear: the Web server process is in command, calling into other processes only as necessary. Developers must carefully write the shared state-holding processes such that they never block because they must multiplex numerous concurrent requests, but no such limitation exists for the Web server process. The clear thread of control means the overall application is easier to develop and debug. The shared-nothing approach underlying this ideal goal and these alternatives is, of course, not specific to Erlang.

As I mentioned earlier, experienced Erlang developers are well aware of potential process bottlenecks such as this one, but I've seen this problem trip up a number of developers new to Erlang. Perhaps the relative simplicity of Erlang concurrency capabilities along with its fast interprocess communication facilities lulls newcomers into a false sense of performance and scalability security when it comes to working with multiple processes. Not knowing the inner workings of OTP behaviors certainly contributes, and another potential point of confusion might be due to erroneously equating modules with processes when designing application logic. But, as always, there's no magic; making the most of concurrency in server-side Web applications means understanding all potential points of resource contention and working to either reduce their effects or, better yet, eliminate them entirely.

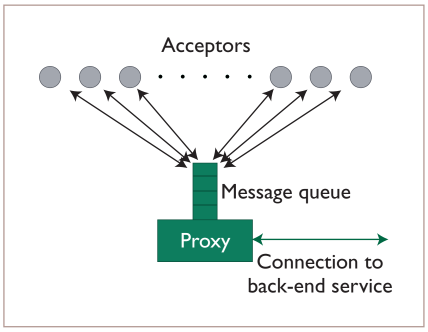

Figure 6. The acceptor processes can overrun the proxy process, depending on the number of concurrent Web clients. When processing a client connection, an acceptor sends a message to the proxy process message queue; for each message, the proxy uses its connection to communicate with the back-end service. The connection to the back-end service is held in the proxy process state. The fact that all acceptors share the proxy resource results in contention, high latency, and possible message queue growth that can lead to out-of-memory conditions.

Acknowledgments

Thanks to Ulf Wiger of Erlang Solutions for his helpful review comments on a draft of this column.

Steve Vinoski is a member of Verivue's technical staff. He's a senior member of IEEE and a member of the ACM. You can read his blog at http://steve.vinoski.net/blog and contact him at vinoski@ieee.org.