SAMPLE CHAPTER

Erlang AND OTP IN ACTION

Martin Logan
Eric Merritt
Richard Carlsson

FOREWORD BY Ulf Wiger

MANNING

Erlang and OTP in Action

by Martin Logan
Eric Merritt
Richard Carlsson

Chapter 3

brief contents

3 Writing a TCP-based RPC service

This chapter covers

- Introduction to OTP behaviors
- Module layout conventions and EDoc annotations
- Implementing an RPC server using TCP/IP
- Talking to your server over telnet

What!? No "hello world"?

That's right, no "hello world." In chapter 2, we provided a review of the Erlang language, and now it's time to do something concrete. In the spirit of getting down and dirty with real-world Erlang, we say *no* to "hello world"! Instead, you'll create something that is immediately useable. You're going to build a TCP-enabled RPC server!

In case you don't know what that is, let us explain. RPC stands for *remote procedure call*. An RPC server allows you to call procedures (that is, functions) remotely from another machine. The TCP-enabled RPC server will allow a person to connect to a running Erlang node, run Erlang commands, and inspect the results with no more than a simple TCP client, like good old Telnet. The TCP RPC server will be a nice first step toward making your software accessible for post-production diagnostics.

Source code

The code for this book is available online at GitHub. You can find it by visiting http://github.com/ and entering "Erlang and OTP in Action" in the search field. You can either clone the repository using `git` or download the sources as a zip archive.

This RPC application, as written, would constitute a security hole if included in a running production server because it would allow access to run any code on that server, but it wouldn't be difficult to limit the modules or functions that this utility could access in order to close that hole. But you won't do that in this chapter. We use the creation of this basic service as a vehicle for explaining the most fundamental, most powerful, and most frequently used of the OTP *behaviours*: the generic server, or `gen_server`. (We stick to the British spelling of *behaviour*, because that's what the Erlang/OTP documentation uses.) OTP behaviours greatly enhance the overall stability, readability, and functionality of software built on top of them.

In this chapter, we cover implementing your first behaviour, and you'll learn about basic TCP socket usage with the `gen_tcp` module (which isn't a behaviour, despite the name). This book is mainly intended for intermediate-level Erlang programmers, and so we start in the thick of it. You'll need to pay strict attention, but we promise the chapter will be gentle enough to fully understand. When you're done, you'll have taken a great leap forward in terms of being able to create reliable software.

By the end of this chapter, you'll have a working Erlang program that will eventually be a part of a production-quality service. In chapter 4, you'll hook this program deeper into the OTP framework, making it an Erlang application that can be composed with other Erlang applications to form a complete deliverable system (also known as a *release*). Much later, in chapter 11, you'll integrate a similar server into the simple cache application that you'll also build. That will be a more robust and scalable version of this TCP server and will show a few interesting twists on the subject. For now, though, let's get more explicit about what you'll be building in this chapter.

3.1 What you're creating

The RPC server will allow you to listen on a TCP socket and accept a single connection from an outside TCP client. After it's connected, it will let a client run functions via a simple ASCII text protocol over TCP. Figure 3.1 illustrates the design and function of the RPC server.

The figure shows two processes. One is the supervisor, as we described in chapter 1; it spawns a worker process that is the actual RPC server. This second process creates a listening TCP socket and waits for someone to

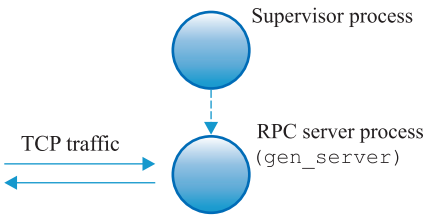

Figure 3.1 RPC server process connected through a socket to the world outside. It accepts requests over TCP, performs them, and returns the results to the client.

connect. When it receives a connection, it reads ASCII text in the shape of normal Erlang function calls, and it executes those calls and returns the result back over the TCP stream. This kind of functionality is useful for any number of things, including remote administration and diagnostics in a pinch. Again, the RPC server understands a basic text protocol over the TCP stream, which looks like standard Erlang function calls. The generic format for this protocol is

```
Module:Function(Arg1, ..., ArgN).
```

For example:

```
lists:append("Hello", "Dolly").
```

(Note that the period character is required.) To interpret these requests, the RPC server parses the ASCII text and extracts the module name, function name, and arguments, transforming them into valid Erlang terms. It then executes the function call as requested and finally returns the results as Erlang terms formatted as ASCII text back over the TCP stream.

Accomplishing this will require an understanding of a number of fundamental Erlang/OTP concepts, all of which we get into in the next couple of sections.

3.1.1 A reminder of the fundamentals

You should already have a basic grasp of modules, functions, messaging, and processes, because we addressed these concepts in chapters 1 and 2. We cover them again here, before we introduce the new concept of behaviours. First, in Erlang, functions are housed in modules, and processes are spawned around function calls. Processes communicate with each other by sending messages. Figure 3.2 illustrates these relationships.

Let's take a second to review these concepts:

- *Modules*—Modules are containers for code. They guard access to functions by either making them private or exporting them for public use. There can be only one module per object file (.beam file). If a module is named `test`, it must reside in a source file called test.erl and be compiled to an object file called test.beam.
- *Functions*—Functions do all the work; all Erlang code in a module must be part of a function. They're the sequential part of Erlang. Functions are executed by processes, which represent the concurrent part. A function must belong to some module.

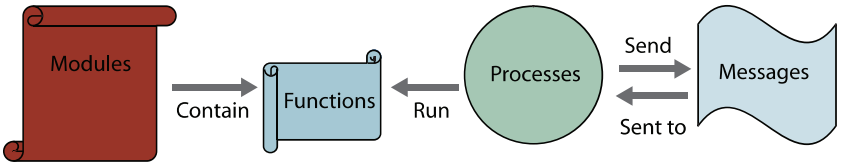

Figure 3.2 The relationships between modules, functions, processes, and messages

- *Processes*—Processes are the fundamental units of concurrency in Erlang. They communicate with each other through messages. Processes are also the basic containers for program state in Erlang: data that needs to be modified over time can be kept inside a process. Any process can create (spawn) another process, specifying what function call it should perform. The new process executes that call and terminates itself when it has finished. A process spawned to perform a simple call to `io:format/2` will be short-lived, whereas one spawned to execute a call like `timer:sleep(infinity)` will last forever, or until someone else kills it.
- *Messages*—Messages are how processes interact. A message can be any Erlang data. Messages are sent asynchronously from one process to another, and the receiver always gets a separate copy of the message. Messages are stored in the mailbox of the receiving process on arrival and can be retrieved by executing a `receive`-expression.

After that quick refresher, let's move on to the concept of behaviours.

3.1.2 Behaviour basics

Behaviours are a way of formalizing common patterns in process-oriented programming. For example, the concept of a server is general and includes a large portion of all processes you'll ever need to write. All those processes have a lot in common—in particular, whether they should be made to follow OTP conventions for supervision and other things. Rewriting all that code for every new server-like process you need would be pointless, and it would introduce minor bugs and subtle differences all over the place.

Instead, an OTP behaviour takes such a recurring pattern and divides it into two halves: the generic part and the application-specific implementation part. These communicate via a simple, well-defined interface. For example, the module you'll create in this chapter will contain the implementation part of the most common and useful kind of OTP behaviour: a generic server, or `gen_server`.

COMPONENTS OF A BEHAVIOUR

In daily use, the word *behaviour* has become rather overloaded and can refer to any of the following separate parts:

- The behaviour interface
- The behaviour implementation
- The behaviour container

The behaviour *interface* is a specific set of functions and associated calling conventions. The `gen_server` behaviour interface contains six functions; `init/1`, `handle_call/3`, `handle_cast/2`, `handle_info/2`, `terminate/2`, and `code_change/3`.

The *implementation* is the application-specific code that the programmer provides. A behaviour implementation is a callback module that exports the functions required by the interface. The implementation module should also contain an attribute

`-behaviour(...).` that indicates the name of the behaviour it implements; this allows the compiler to check that the module exports all the functions of the interface. The listing that follows shows those parts of the module header and the interface functions that must be implemented for a valid `gen_server`.

Listing 3.1 Minimal `gen_server` behaviour implementation module

```
-module(...).

-behaviour(gen_server).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-record(state, {}).

init([]) ->
    {ok, #state{}}.

handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

If any of these functions are missing, the behaviour implementation isn't fully conforming to the `gen_server` interface, in which case the compiler issues a warning. We get into detail about what each of these functions do in the next section, when you implement them in order to build your RPC server.

The third and final part of a behaviour is the *container*. This is a process that runs code from a library module and that uses implementation callback modules to handle application-specific things. (Technically, the container could consist of multiple processes working closely together, but usually there is only one process.) The name of the library module is the same as that of the behaviour. It contains the generic code for that behaviour, including functions to start new containers. For example, for a `gen_server` behaviour, the code sits within the `gen_server` module that can be found in the `stdlib` section of the Erlang/OTP libraries. When you call `gen_server:start(...,foo,...)`, a new `gen_server` container is created that uses `foo` as a callback module.

Behaviour containers handle much of what is challenging about writing canonical, concurrent, fault-tolerant OTP code. The library code takes care of things like synchronous messaging, process initialization, and process cleanup and termination, and also provides hooks into larger OTP patterns and structures like code change and supervision trees.

Containers

The word *container* as used here is our own choice of terminology, but we find it fitting. The OTP documentation tends to talk only about the process, but that doesn't convey the division of responsibility in a behaviour and can be unclear at times. (If you have some familiarity with J2EE containers in Java, there are many similarities here, but also some differences: an OTP container is lightweight, and the container is the only real object in this context.)

Instantiating a behaviour

The whole point of a behaviour is to provide a template for processes of a particular type. Every behaviour library module has one or more API functions (generally called `start` and/or `start_link`) for starting a new container process. We call this *instantiating* the behaviour.

Process type

The informal notion of process *type* (regardless of whether behaviours are involved) lets us talk about things like a `gen_server` process. Processes are of the same type if they're running mainly the same code, which means that they understand mainly the same kind of messages. The only difference between two processes of the same type is their individual state. Processes of the same type generally have the same *spawn signature* or *initial call*; that is to say, they had the same function as starting point.

In some cases, you'll write a behaviour implementation module so that there can only be one instance at a time; in other cases, you may want to have thousands of simultaneous instances that all run the same code but with different data. The important thing to keep in mind is that when your callback code is running, it's executed by a container, which is a process with identity and state (including its mailbox). This is a lot like objects in object-oriented programming, but with the addition that all these containers are living things that are running code in parallel.

To summarize, the behaviour interface is the contract that allows the behaviour implementation (your code) to leverage the power of the behaviour container. The purpose is to make it simple to implement processes that follow typical concurrent programming patterns. Working with OTP behaviours has a number of advantages:

- Developers get more done with less code—sometimes much less.
- The code is solid and reliable because it has well-tested library code at its core.
- The code fits into the larger OTP framework, which provides powerful features such as supervision for free.
- The code is easier to understand because it follows a well-known pattern.

With this basic understanding of behaviours, we can now move on to the implementation of the RPC server, which will utilize all of what we've described. Everything you do from here on is related to implementing the TCP RPC server. This exercise will cover a lot. At one level, it's about how to use behaviours. You'll be coding up a behaviour implementation that conforms to a behaviour interface, and you'll see how the `gen_server` behaviour provides all the functionality you're looking for. At another level, what you'll be doing here is even more fundamental: starting to use Erlang within the framework of OTP.

3.2 Implementing the RPC server

If you're an intermediate-level Erlang programmer, you already have some familiarity with modules, processes, functions, and messaging. But it's likely that your experience is from a more informal, plain Erlang context. We revisit these concepts in this chapter in the context of OTP. If you're new to Erlang and this is your first book, you're probably an experienced programmer from a different background. In that case, you don't need any prior knowledge of these things to grasp what we cover in this chapter.

It's our opinion that writing pure Erlang code with processes and message passing (and getting everything right) *without* OTP is an advanced topic and is something you should resort to only when you must. Perhaps not having done this sort of programming in pure Erlang is a blessing, because you'll pick up the right OTP habits straight away—maybe even the strict approach we take to module structure and layout, inline documentation, and commenting.

Because you'll need a module to contain your behaviour implementation, we start with a little about module creation and layout.

3.2.1 Canonical module layout for a behaviour implementation

One of the nice things about behaviours is that they give you a great amount of consistency. When looking at a behaviour implementation module, you'll recognize aspects that are common to all such modules, like the behaviour interface functions and the customary `start` or `start_link` function. To make the files even more recognizable, you can adopt the canonical behaviour implementation module layout that we elaborate on here.

This standard layout consists of four sections. Table 3.1 details them in the order that they appear in the file.

Table 3.1 Source code sections of a canonical behaviour implementation module

Section	Description	Functions exported	EDoc annotations
Header	Module attributes and boilerplate	N/A	Yes, file level
API	Programmer interface; how the world interacts with the module	Yes	Yes, function level

Table 3.1 Source code sections of a canonical behaviour implementation module *(continued)*

Section	Description	Functions exported	EDoc annotations
Behaviour interface	Callback functions required by the behaviour interface	Yes	Optional
Internal functions	Helper functions for the API and behaviour interface functions	No	Optional

We'll now look at the details of implementing each of these sections in turn, starting with the module header.

3.2.2 The module header

Before you can create the header, you need to create a file to hold it. Because you'll be building a TCP-based RPC server, let's create a file named tr_server.erl where you'll place this code. Use your favorite text editor.

> **Module naming conventions and the flat namespace**
>
> Erlang has a flat namespace for modules. This means module names can collide. (There exists an experimental Java-like package system in Erlang, but it hasn't caught on and isn't fully supported.) If modules are given names like `server`, it's easy to end up with two modules from different projects that have the same name. To avoid such clashes, the standard practice is to give module names a suitable prefix. Here, we've taken the first two letters of the acronyms TCP and RPC: `tr_server`.

The first thing you need to enter is the file-level header comment block:

```
%%%-------------------------------------------------------------------
%%% @author Martin & Eric <erlware-dev@googlegroups.com>
%%%  [http://www.erlware.org]
%%% @copyright 2008-2010 Erlware
%%% @doc RPC over TCP server. This module defines a server process that
%%%      listens for incoming TCP connections and allows the user to
%%%      execute RPC commands via that TCP stream.
%%% @end
%%%-------------------------------------------------------------------
```

Note that each comment line begins with three `%` characters, although a single `%` is sufficient. This is a convention used for file-level comments, whose contents apply to the file as a whole. Furthermore, this may be the first time you've seen comments containing EDoc annotations. EDoc is a tool for generating documentation directly from source code annotations (similar to Javadoc) and comes with the standard Erlang/OTP distribution. We don't have room in this book to get too deep into how to use EDoc: you can read more about it in the tools section of the OTP documentation. We'll spend a little time on it here, because it's the de facto standard for in-code

Erlang documentation. We suggest that you familiarize yourself with EDoc and make a habit of using it in your own code.

All EDoc tags begin with an `@` character. Table 3.2 describes the tags in this header. We return to this subject at the end of chapter 4, after we've explained how OTP applications work; there, we show briefly how to run EDoc to generate the documentation.

Table 3.2 Basic EDoc tags

EDoc tag	Description
`@author`	Author information and email address.
`@copyright`	Date and attribution.
`@doc`	General documentation text. First sentence is used as a summary description. Can contain valid XHTML and some wiki markup.
`@end`	Ends any tag above it. Used here so that the line `%%%----...` isn't included in the text for the previous `@doc`.

The first thing in your file that isn't a comment is the `-module(...)` attribute. The name supplied must correspond to the file name; in this case, it looks like

```
-module(tr_server).
```

(Remember that all attributes and function definitions must end with a period.) After the module attribute, the next thing to add is the behaviour attribute. This indicates to the compiler that this module is intended to be an implementation of a particular behaviour and allows it to warn you if you forget to implement and export some behaviour interface functions. You'll implement a generic server, so you want the following behaviour attribute:

```
-behaviour(gen_server).
```

Next come the export declarations. You'll typically have two. (The compiler combines them, but grouping related functions helps readability.) The first is for your API section, and the second is for the behaviour interface functions that must also be exported. Because you haven't yet designed the API, a placeholder will suffice for now. But you know which the behaviour interface functions are, so you can list them right away:

```
%% API
-export([]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).
```

Note the comment above the second declaration. The behaviour interface functions are often referred to as *callbacks*. This is because at startup, the name of the behaviour

implementation module is passed to the new container, which then calls back into the implementation module through these interface functions. We go into more detail about the use of each of the interface functions later in the chapter.

Following the exports, there may be a number of optional application-specific declarations and/or preprocessor definitions. They're highlighted in the following listing, which shows the complete header for the `tr_server` module.

Listing 3.2 Full tr_server.erl header

```
%%%-------------------------------------------------------------------
%%% @author Martin & Eric <erlware-dev@googlegroups.com>
%%%  [http://www.erlware.org]
%%% @copyright 2008 Erlware
%%% @doc RPC over TCP server. This module defines a server process that
%%%      listens for incoming TCP connections and allows the user to
%%%      execute RPC commands via that TCP stream.
%%% @end
%%%-------------------------------------------------------------------
-module(tr_server).

-behaviour(gen_server).

%% API
-export([
         start_link/1,
         start_link/0,
         get_count/0,
         stop/0
         ]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-define(SERVER, ?MODULE).
-define(DEFAULT_PORT, 1055).

-record(state, {port, lsock, request_count = 0}).
```

❶ Sets SERVER to module name

❷ Defines default port

❸ Holds state of process

Macros are commonly used for various constants, to ensure that you only need to modify a single place in the code to change the value (see section 2.12). Here, you use them to define which default port to use ❷ and to set up `SERVER` as an alias for the name of your module ❶ (you may want to change that at some point, so you shouldn't assume that the server name will always remain the same as the module name). After the macros, you define the name and the format of the record (see section 2.11) that will hold the live state of your server process while it's running ❸.

Now that the header is complete, the next section of your behaviour implementation module is the API.

3.2.3 The API section

All the functionality that you want to make available to the users of your module (who don't care much about the details of how you implemented it) is provided through

the application programming interface (API) functions. The main things that a user of a generic server wants to do are

- Start server processes
- Send messages to these processes (and receive the answers)

To help you implement this basic functionality, there are three primary `gen_server` library functions. These are listed in table 3.3.

Table 3.3 `gen_server` library functions for implementing the API

Library function	Associated callback function	Description
`gen_server:start_link/4`	`Module:init/1`	Starts a `gen_server` container process and simultaneously links to it
`gen_server:call/2`	`Module:handle_call/3`	Sends a synchronous message to a `gen_server` process and waits for a reply
`gen_server:cast/2`	`Module:handle_cast/2`	Sends an asynchronous message to a `gen_server` process

Basically, your API functions are simple wrappers around these library calls, hiding such implementation details from your users. The best way to illustrate how these functions work is to use them to implement the API for the `tr_server` module, as shown in the following listing.

Listing 3.3 API section of tr_server.erl

```
%%%===================================================================
%%% API                                                          <-- Banner at start of section
%%%===================================================================

%%--------------------------------------------------------------------
%% @doc Starts the server.
%%
%% @spec start_link(Port::integer()) -> {ok, Pid}
%% where
%%  Pid = pid()
%% @end
%%--------------------------------------------------------------------
start_link(Port) ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [Port], []).   <-- Spawns server process

%% @spec start_link() -> {ok, Pid}
%% @doc Calls `start_link(Port)' using the default port.
start_link() ->
    start_link(?DEFAULT_PORT).
```

```
%%--------------------------------------------------------------------
%% @doc Fetches the number of requests made to this server.
%% @spec get_count() -> {ok, Count}
%% where
%%  Count = integer()
%% @end
%%--------------------------------------------------------------------
get_count() ->
    gen_server:call(?SERVER, get_count).          <-- 1 Makes caller wait for reply

%%--------------------------------------------------------------------
%% @doc Stops the server.
%% @spec stop() -> ok
%% @end
%%--------------------------------------------------------------------
stop() ->
    gen_server:cast(?SERVER, stop).               <-- 2 Doesn't wait for reply
```

A query like ❶ uses `gen_server:call/2`, which makes the caller wait for a reply. A simple command like `stop` ❷ typically uses the asynchronous `gen_server:cast/2`.

> **Only one process type per module**
>
> The same module may be used to spawn many simultaneous processes but should contain code for only one type of process (apart from the API code, which by definition is executed by the clients, who could be of any type). If various parts of the code in a single module are meant to be executed by different types of processes, it becomes hard to reason about the contents of the module, as well as about the system as a whole, because the role of the module isn't clear.

Briefly, the API in listing 3.3 tells you that a `tr_server` can do three things:

- It can be started using `start_link()` or `start_link(Port)`.
- It can be queried for the number of requests it has processed, using `get_count()`.
- It can be terminated by calling `stop()`.

Before we get into the details of how these functions work, and the communication between the caller and the server process (the container), let's refresh your memory with regard to messaging in Erlang.

Quick reminder about processes and communication

Processes are the building blocks of any concurrent program. Processes communicate via messages that are posted asynchronously; on arrival, they're queued up in the mailbox of the receiving process. Figure 3.3 illustrates how messages enter the process mailbox, where they're kept until the receiver decides to look at them.

This ability of processes to automatically buffer incoming messages and selectively handle only those messages that are currently relevant is a crucial feature of Erlang (see section 2.13.2 for more about selective receive).

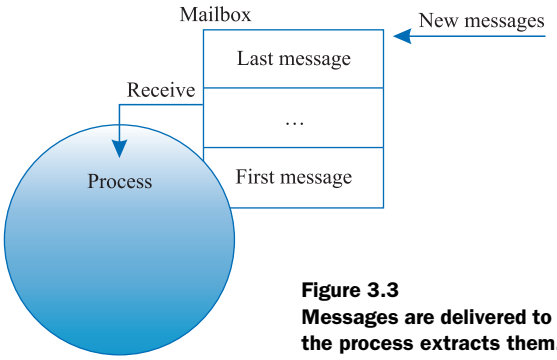

Figure 3.3
Messages are delivered to the mailbox of a process and stay there until the process extracts them. There is no size limit on the mailbox.

With this in mind, we now look at how the OTP libraries take away a lot of the fiddly details of message passing between clients and servers, instead handing you a set of higher-level tools for process communication. These tools may not be as supremely flexible as hacking your own communication patterns, but they're solid, straightforward, and fit most everyday situations. They also guarantee several important properties like timeouts, supervision, and error handling, which you would otherwise have to code manually (which can be boring, verbose, and hard to get completely right).

HIDING THE PROTOCOL

The set of messages that a process will accept is referred to as its *protocol*. But you don't want to expose the details of these messages to your users, so one of the main tasks of the API is to hide this protocol from the rest of the world.

Your `tr_server` process will accept the following simple messages:

- `get_count`
- `stop`

These are plain atoms, but there's no need for users of the `tr_server` module to know this implementation detail; you'll keep all that hidden behind the API functions. Imagine a future extension of your server that requires users to log in before they're allowed to send requests. Your API might then need a function to create a user on the server, which could look something like this:

```
add_user(Name, Password, Permissions) ->
  gen_server:call(?SERVER, {add_user, [{name, Name},
                                       {passwd, Password},
                                       {perms, Permissions}]}).
```

Such a complex message format is something you don't want to leak out of your module; you might want to change it in the future, which would be hard if clients were depending on it. By wrapping the communication with the server in an API function, the users of your module remain oblivious to the format of these messages.

Finally, on a primitive level, all messages in Erlang are sent asynchronously (using the `!` operator), but in practice you often have to block when you can't do

Double blind

Another level of hiding is going on here: the OTP libraries are hiding from you the details of the real messages going back and forth between processes. The message data that you pass as arguments to `call/2` and `cast/2` is only the payload. It's automatically wrapped up along with a bit of metadata that allows the `gen_server` container to see what kind of message it is (so it knows which callback should handle the payload) and to reply to the right process afterwards.

anything useful before some expected answer arrives. The `gen_server:call/2` function implements this synchronous request-reply functionality in a reliable way, with a default timeout of 5 seconds before it gives up on getting the answer (in addition, the version `gen_server:call/3` lets you specify a timeout in milliseconds, or `infinity`).

Now that we've explained the purpose behind the API functions and the `gen_server` library functions used to implement them, we can get back to the code.

API FUNCTIONS IN DETAIL

Listing 3.3 showed the implementation of the API, and it's time that we explain exactly what it does. First, table 3.4 summarizes the four API functions.

Singleton process

To keep things simple, we've designed this particular server to be a singleton: you can have only one instance running at a time. When it starts, the new `gen_server` container process is registered under the name specified by the `SERVER` macro defined in listing 3.2 (that's what the argument `{local, ?SERVER}` means in listing 3.3). This makes it possible for the functions `get_count()` and `stop()` to communicate with it by name. If you want to run several server instances simultaneously, you must modify the program a bit, because processes can't have the same registered name (see section 2.13.3 for details about the process registry). On the other hand, it's common to have servers that provide a system-level service, of which there can be only one per Erlang node (or even one per Erlang cluster); so, a singleton server like this isn't an unrealistic example.

Table 3.4 The `tr_server` API

API function	Description
`start_link/1`	Starts the `tr_server` listening on a specific port
`start_link/0`	Alias for `start_link/1`, using the default port
`get_count/0`	Returns the number of requests processed so far
`stop/0`	Shuts down the server

This is how the API functions work:

- `start_link(Port)` and `start_link()`—Start your server process and at the same time link to it. This is done by calling `gen_server:start_link/4` and is where you indicate (in the second argument) to the behaviour container which callback module contains the implementation to be used. The normal thing to do is to pass the value of the built-in macro `MODULE`, which always expands to the name of the current module (see section 2.12.1):

  ```
  gen_server:start_link({local, ?SERVER}, ?MODULE, [Port], [])
  ```

  When this call is executed, it spawns a new `gen_server` container process, registers it on the local node using the name that the `SERVER` macro expands to, and waits until it has been initialized by running the `init/1` callback function of the implementation module (more on this in section 3.2.4) before returning to the caller. At that point, the server is up and running, fully initialized and ready to accept messages.

  The third argument, in this case `[Port]`, provides data to the server on startup. This is passed as is to the `init/1` callback function, where you can use it to set up the initial process state. The fourth argument is a list of extra options, which you'll leave empty for now. Note that from the API user's point of view, all these details are hidden; there is only a single argument: the port that the server should listen on.

- `get_count()`—Uses `gen_server:call/2` to send the atom `get_count` as a synchronous request to the server. This means the call waits for the reply from the server, temporarily suspending the calling process:

  ```
  gen_server:call(?SERVER, get_count)
  ```

  The first argument in the call is either the registered name or the process ID of the server process; here, you use the same name (the `SERVER` macro) that was used to register the process in the `start_link/1` function. The second argument in the call is the message to be sent. When the server has received and handled this message, it sends a reply back to the calling process. The `gen_server:call/2` function takes care of receiving this reply and returning it as the result from the function call, so the caller doesn't need to know anything about how to send or receive messages.

  Also note that the atom `get_count` used in the message (as part of the server protocol) has the same name as the API function; this is helpful when you're reading the code or debugging—don't make the internal server protocol cryptic just because you can.

- `stop()`—Uses `gen_server:cast/2` to send the atom `stop` as an asynchronous message (meaning that the function returns immediately without waiting for a reply):

  ```
  gen_server:cast(?SERVER, stop)
  ```

After you've sent this message, you assume that the container will shut itself down as soon as it receives the message. You don't need a reply; hence `cast`, rather than `call`.

That's all the functions you need for this simple server. After all, most of its real functionality will be provided via the TCP connection, so these API functions are only needed for starting, stopping, and checking the status.

THE @SPEC TAG

Before we move on to define the behaviour interface callback functions, we want to explain briefly the new EDoc tag you used in the documentation before each function (listing 3.3). It's highly recommended that you have at least a `@doc` annotation for each and every API function, as well as for the module as a whole (listing 3.2). The additional `@spec` tag can be used to describe the type of data that the function accepts as input and what type of values it can return. For example, the `@spec` for `start_link/1`

```
%% @spec start_link(Port::integer()) -> {ok, Pid}
%% where
%%  Pid = pid()
```

indicates that the function takes a single argument that is an integer and returns a tuple `{ok, Pid}`, where `Pid` is a process identifier. Type names always look like function calls, as in `integer()`, so that they aren't confused with atoms. Types can be attached directly to variables with the `::` notation as with `Port`, or they can be listed at the end of the specification as with `where Pid = ...`.

You're finished with the user API section, so it's finally time to begin implementing the behaviour interface functions—the callbacks, where most of the real work is done.

3.2.4 The callback function section

Each of the `gen_server` library functions you use in your API corresponds to a specific callback function specified by the `gen_server` behaviour interface. These callbacks now need to be implemented. To refresh your memory, table 3.5 repeats table 3.3, with the addition of `handle_info/2`, which doesn't correspond to any of the library functions used for the API.

First, look back at the `tr_server:start_link/1` function in listing 3.3. That function hides that fact that you're calling `gen_server:start_link/4`; and as you can see from table 3.5, the new container then calls back to `tr_server:init/1` (which must be exported by the `tr_server` module, as required by the `gen_server` behaviour interface) in order to perform the initialization. Similarly, `tr_server:get_count/0` shields the user from having to worry about your protocol and the fact that the communication is performed by `gen_server:call/2`. When such a message is received by the container, it calls back to `tr_server:handle_call/2` in order to handle the message; in this case, the only possible message of this kind is the atom `get_count`. Analogously, `tr_server:stop/0` uses `gen_server:cast/2` to dispatch a message to the

Table 3.5 `gen_server` library functions and callbacks

Library function	Associated callback function	Description
`gen_server:start_link/4`	`Module:init/1`	Starts a `gen_server` container and simultaneously links to it.
`gen_server:call/2`	`Module:handle_call/3`	Sends a synchronous message to a `gen_server` container and waits for a reply.
`gen_server:cast/2`	`Module:handle_cast/2`	Sends an asynchronous message to a `gen_server` container.
N/A	`Module:handle_info/2`	Handles messages sent to a `gen_server` container that were not sent using one of the `call` or `cast` functions. This is for out-of-band messages.

container asynchronously; and on receiving such a message, the container calls back to `tr_server:handle_cast/2`.

But notice that the `handle_info/2` callback doesn't correspond to any `gen_server` library function. This callback is an important special case. It's called to handle any messages that arrive in the mailbox of a `gen_server` that weren't sent using one of the `call` or `cast` library functions (typically, naked messages sent with the plain old `!` operator). There can be various reasons for such messages to find their way to the mailbox of a `gen_server` container—for example, that the callback code requested some data from a third party. In the case of your RPC server, you'll receive data over TCP, which will be pulled off the socket and sent to your server process as plain messages.

After all this talk, the next listing shows what your callback functions do.

Listing 3.4 `gen_server` callback section for `tr_server`

```
%%%===================================================================
%%% gen_server callbacks
%%%===================================================================

init([Port]) ->
    {ok, LSock} = gen_tcp:listen(Port, [{active, true}]),
    {ok, #state{port = Port, lsock = LSock}, 0}.

handle_call(get_count, _From, State) ->
    {reply, {ok, State#state.request_count}, State}.

handle_cast(stop, State) ->
    {stop, normal, State}.
```

1 Initializes server

2 Returns request count

3 Shuts down gen_server

Upon initialization of the server, the `init` function creates a TCP listening socket, sets up the initial state record, and also signals an immediate timeout. Next, the code

returns the current request count to the calling client process. A special return value `stop` tells the `gen_server` process to shut down.

As you can see in listing 3.4, the first three callback functions are almost trivial. (We leave `handle_info/2` and the other two for later, in listing 3.5.) The most complicated thing about these three functions is the format of the values they return to communicate back to the `gen_server` container. Let's go through them in detail:

- `init/1`, *initialization callback*—This function is called whenever you start a new `gen_server` container, for example, via `gen_server:start_link/4`. These are the first examples of how OTP helps you write industrial-strength code with a minimum of effort. The `start_link` library function sets you up to be hooked into the powerful process-supervision structures of OTP. It also provides critical initialization functionality, blocking the caller until the process is up and running and registered (if requested), and the `init/1` callback has completed. This ensures that your process is fully operational before it starts to process requests.

  Breaking this function down line by line, the first thing you see is `init([Port]) ->`, meaning that `init` takes one argument, which must be a list containing a single element that you call `Port`. Note that this matches exactly what you passed from the `start_link/1` function in listing 3.3. (Always passing a list, even with a single element, is a common convention for `init/1`.)

  Next, you create your TCP listening socket on the specified port ❶, using the standard library `gen_tcp` module:

  ```
  {ok, LSock} = gen_tcp:listen(Port, [{active, true}]),
  ```

  A *listening socket* is a socket that you create and wait on to accept incoming TCP connections. After you accept a connection, you have an active socket from which you can receive TCP datagrams. You pass the option `{active, true}`, which tells `gen_tcp` to send any incoming TCP data directly to your process as messages.

  Last, you return from `init/1` with a 3-tuple containing the atom `ok`, your process state (in the form of a `#state{}` record), and a curious `0` at the end:

  ```
  {ok, #state{port = Port, lsock = LSock}, 0}.
  ```

  The `0` is a timeout value. A timeout of zero says to the `gen_server` container that immediately after `init/1` has finished, a timeout should be triggered that forces you to handle a timeout message (in `handle_info/2`) as the first thing you do after initialization. The reason for this will be explained shortly.

- `handle_call/3`, *callback for synchronous requests*—This function is invoked every time a message is received that was sent using `gen_server:call/2`. It takes three arguments: the message (as it was passed to `call`), `From` (let's not worry about that yet), and the current state of the server (which is whatever you want it to be, as set up in `init/1`).

You have a single synchronous message to handle: `get_count`. And all you need to do is extract the current request count from the state record and return it. As earlier, the return value is a 3-tuple ❷, but with slightly different content than in `init/1`:

```
{reply, {ok, State#state.request_count}, State}.
```

This indicates to the `gen_server` container that you want to send a reply to the caller (you should, because it's expected); that the value returned to the caller should be a tuple `{ok, N}`, where `N` is the current number of requests; and finally that the new state of the server should be the same as the old (nothing was changed).

- `handle_cast/2`, *callback for asynchronous messages*—Your API function `stop()` uses `gen_server:cast/2` to dispatch an asynchronous message `stop` to the server, without waiting for any response. Your task is to make your server terminate when it receives this message. Any message sent using `cast` is handled by the `tr_server:handle_cast/2` callback; this is similar to `handle_call/3`, except that there is no `From` argument. When your `handle_cast` function sees the message `stop`, it only has to return the following 3-tuple:

  ```
  {stop, normal, State}.
  ```

  This tells the `gen_server` container that it should stop ❸ (that is, terminate), and that the reason for termination is `normal`, which indicates a graceful shutdown. The current state is also passed on unchanged (even though it won't be used further). Note here that the atom `stop` returned in this tuple instructs the container to shut down, whereas the `stop` message used in the protocol between the API and the server could have been any atom (such as `quit`), but was chosen to match the name of the API function `stop()`.

By now, we've covered most of the important points regarding the `gen_server` behaviour: the interface, the callback functions, the container, and how they interact. There is certainly more to learn about `gen_server`, but we return to that throughout the book. There is one thing left to discuss regarding the server you implement here: handling out-of-band messages. In many typical servers, there are no such messages to handle; but in this particular application, it's where you do all the heavy lifting.

HANDLING OUT-OF-BAND MESSAGES

As we explained, any messages to a `gen_server` process that weren't sent using `call` or `cast` are handled by the `handle_info/2` callback function. These are considered *out-of-band messages* and can happen when your server needs to communicate with some other component that relies on direct messages rather than on OTP library calls—for example, a socket or a port driver. But you should avoid sending out-of-band messages to a `gen_server` if you can help it.

In the `init/1` function, you set up a TCP listening socket for the server, and then you mysteriously return a timeout value of `0` from that function, which you know will trigger an immediate timeout (see listing 3.4).

`gen_server` timeout events

When a `gen_server` has set a timeout, and that timeout triggers, an out-of-band message with the single atom `timeout` is generated, and the `handle_info/2` callback is invoked to handle it. This mechanism is usually used to make servers wake up and take some action if they have received no requests within the timeout period.

Here, you're abusing this timeout mechanism slightly (it's a well-known trick) to allow the `init/1` function to finish quickly so that the caller of `start_link(...)` isn't left hanging; but at the same time, you're making sure the server immediately jumps to a specific piece of code (the `timeout` clause of `handle_info/2`) where it can get on with the more time-consuming part of the startup procedure—in this case, waiting for a connection on the socket you created (see end of listing 3.5). Because you're not using server timeouts for anything else in this application, you know it won't return to that point again afterward.

But back to TCP sockets: an active socket like this forwards all incoming data as messages to the process that created it. (With a passive socket, you'd have to keep asking it if there is more data available.) All you need to do is to handle those messages as they're arriving. Because they're out-of-band data as far as the `gen_server` container is concerned, they're delegated to the `handle_info/2` callback function, shown in listing 3.5.

Listing 3.5 `handle_info/2`, `terminate/2`, and `code_change/3` callback functions

```
handle_info({tcp, Socket, RawData}, State) ->
    do_rpc(Socket, RawData),
    RequestCount = State#state.request_count,
    {noreply, State#state{request_count = RequestCount + 1}};
handle_info(timeout, #state{lsock = LSock} = State) ->
    {ok, _Sock} = gen_tcp:accept(LSock),
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

Increments request count after RPC requests

Obligatory but uninteresting for now

Let's go through this function like we did with the three previous callbacks. `handle_info/2` is the callback for out-of-band messages. This function has two clauses; one for incoming TCP data and one for the timeout. The timeout clause is the simplest, and as we explained earlier, it's also the first thing the server does after it has finished running the `init/1` function (because `init/1` set the server timeout to zero): a kind of deferred initialization. All this clause does is use `gen_tcp:accept/1` to wait for a TCP connection on your listening socket (and the server will be stuck here until that happens). After a connection is made, the timeout clause returns and signals to the `gen_server` container that you want to continue as normal with an unchanged state.

(You don't need to remember the socket handle returned by `accept`, because it's also included in each data package.)

That finally brings you to the clause that matches messages on the form `{tcp, Socket, RawData}`. This is the kind of message that an active socket sends to its owner when it has pulled data off the TCP buffer. The `RawData` field is what you're interested in; it's the ASCII text that the client has sent to you. (You're finally getting back to the purpose of the program: to handle RPC requests over TCP!) The bulk of the code is in a helper function `do_rpc/2` that's shown in listing 3.6; all you need to do here after the RPC has been executed is to update the request count in the server state (see section 2.11.3 for details about updating record fields) and return control to the `gen_server` container.

THE INTERNAL FUNCTIONS

If you've come this far in the chapter, you're excused if you aren't too interested in how you implement the `do_rpc/2` function. In that case, you can skip directly to section 3.3 (perhaps stopping to type in the code from listing 3.6 first) and look at how you run this server and talk to it over TCP. But if you want to see a few techniques for handling input, parsing, and performing meta-calls, keep reading.

Listing 3.6 Internal functions

```
%%%==========================================================================
%%% Internal functions
%%%==========================================================================

do_rpc(Socket, RawData) ->
    try
        {M, F, A} = split_out_mfa(RawData),
        Result = apply(M, F, A),
        gen_tcp:send(Socket, io_lib:fwrite("~p~n", [Result]))
    catch
        _Class:Err ->
            gen_tcp:send(Socket, io_lib:fwrite("~p~n", [Err]))
    end.

split_out_mfa(RawData) ->
    MFA = re:replace(RawData, "\r\n$", "", [{return, list}]),
    {match, [M, F, A]} =
        re:run(MFA,
               "(.*):(.*)\s*\\((.*)\s*\\)\s*.\s*$",
               [{capture, [1,2,3], list}, ungreedy]),
    {list_to_atom(M), list_to_atom(F), args_to_terms(A)}.

args_to_terms(RawArgs) ->
    {ok, Toks, _Line} = erl_scan:string("[" ++ RawArgs ++ "]. ", 1),
    {ok, Args} = erl_parse:parse_term(Toks),
    Args.
```

1 Performs requested call

2 Outputs result

3 Strips CRLF

4 Parses request string

To give a quick overview, the code in listing 3.6 has four main parts: splitting the input, parsing the function arguments, executing the requested call, and sending back the result. First, notice that the inner workings of the `do_rpc/2` function are

Check the borders

Checking data as it passes from the untrusted world into the trusted inner sanctum of your code is a fundamental design principle of Erlang programs. After you verify that the data conforms to your expectations, there is no need to check it repeatedly: you can code for the correct case and let supervision take care of the rest. The reduction in code size from using this technique can be significant, and so can the reduction in number of programming errors, due to the improved readability. Any remaining errors, because you aren't masking them, show up as process restarts in your logs, which allows you to correct the problems as they occur. Let it crash!

wrapped in a `try` expression (section 2.8.2). Because you're working on data from the outside world, several things could go wrong, and this is an easy way to ensure that if the code crashes (throws an exception), you print the error message and continue rather than crashing the entire server process. On the other hand, this doesn't protect against correct but malignant requests, as you'll see in section 3.3.

First you use the standard library `re` module (Perl-compatible regular expressions) to strip the trailing carriage return and line feed ❸. This should leave only text on the form `Module:Function(Arg1,...,ArgN)`. according to the protocol defined at the start of section 3.1. (Otherwise, you'll crash at some point, and the `try` expression will handle it.)

Next, you use the `re` module again ❹ to extract the `Module`, `Function`, and `Arg1,...,ArgN` parts. The details of using regular expressions are beyond the scope of this book, so check the standard library documentation for more information. The module and function names should have the form of Erlang atoms, so all you need to do is convert them from strings to atoms.

But the arguments could be much more complicated. They're a comma-separated list of terms, and there could be zero, one, or more. You handle them in `args_to_terms/1`, where you use a couple of standard library functions to first tokenize the string (placed within angle brackets to make a list, and ended with a period character) and then parse the tokens to form a real Erlang list of terms.

I/O lists: easy scatter/gather

It's worth noting that the result from `io_lib:fwrite/2` might not be a normal string (that is, a flat list of characters). It can still be passed directly to a socket, though; it's what is known as an *I/O list*: a possibly nested, deep, list that may contain both character codes and chunks of binary data. This way, no intermediate concatenated lists need to be created in order to output a number of smaller I/O lists in sequence: make a list of the segments, and pass the entire thing to the output stream. This is similar to the scatter/gather techniques found in modern operating systems.

The module name, function name, and list of argument terms are then passed to the built-in function `apply/3` ❶. This looks much like `spawn/3` (see section 2.13) but doesn't start a new process—it executes the corresponding function call. (It's what we call a *meta-call* operator.) The value returned from this function call is finally formatted as text by `io_lib:fwrite/2` ❷ and sent back over the socket as the response to the user—a remote procedure call has been performed!

Your RPC server is now done and ready to try out. In the next section, you'll give it a trial run and see if it works.

3.3 Running the RPC server

The first step in getting this running is compiling the code. (As we said at the start of the chapter, the complete source files are available online, at GitHub.com.) Run the command `erlc tr_server.erl`. If it completes without any errors, you have a file named tr_server.beam in your current directory. Start an Erlang shell in the same directory, and start the server, as follows:

```
Eshell V5.6.2  (abort with ^G)
1> tr_server:start_link(1055).
{ok,<0.33.0>}
```

We picked port 1055 arbitrarily here because it's easy to remember (10 = 5 + 5). The call to `start_link` returns a tuple containing `ok` and the process identifier of the new server process (although you don't need that now).

Next, start a telnet session on port 1055. On most systems (not all Windows versions, however—download a free telnet client such as PuTTY if needed), you can do this by entering `telnet localhost 1055` at a system shell prompt (*not* in your Erlang shell). For example:

```
$ telnet localhost 1055
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
init:stop().
ok
Connection closed by foreign host.
```

The first session was a success! Why? Let's inspect the dialog and see exactly what happened.

First, you use telnet to connect via TCP on port 1055 to the running `tr_server`. After you connect, you enter the text `init:stop().`, which is read and parsed by the server. You expect this to result in the server calling `apply(init, stop, [])`. You also know that `init:stop/0` returns the atom `ok`, which is exactly what you see printed as the result of your request. But the next thing you see is "Connection closed by foreign host." This was printed by the telnet utility because the socket it was connected to suddenly closed at the remote end. The reason is that the call to `init:stop()` shut down the entire Erlang node where the RPC server was running. This demonstrates both that your RPC server works *and* just how dangerous it can be to give someone

A server should not call itself

With your RPC server, you can try calling any function exported from any module available on the server side, except one: your own `tr_server:get_count/0`. In general, a server can't call its own API functions. Suppose you make a synchronous call to the same server from within one of the callback functions: for example, if `handle_info/2` tries to use the `get_count/0` API function. It will then perform a `gen_server:call(...)` to itself. But that request will be queued up until after the current call to `handle_info/2` has finished, resulting in a circular wait—the server is deadlocked.

unrestricted access to run code on your node! In an improved version, you might want to limit which functions a user can call and even make that configurable.

To conclude, in not that many lines of code, you've built an application that can (with a bit of tweaking) be useful in the real world. And more important, it's a stable application that fits into the OTP framework.

3.4 A few words on testing

Before you're done, there is one more thing that you as a conscientious developer should do: add some testing. Many would argue that you should have started with testing, writing your tests first to guide your development. But adding test code to the examples would clutter the code and distract from the main points we're trying to make; this is, after all, a book about the OTP framework, not about testing. Also, the art of writing tests for concurrent, distributed systems like the ones you're creating here could easily be the subject of a book or two.

Two levels of testing are of immediate interest to a developer: unit testing and integration testing. Unit testing is focused on creating tests that are ready to run at the press of a button and that test specific properties of a program (preferably, at most one property per test). Integration testing is more about testing that a number of separately developed components work together and may require some manual work to set up everything before the tests can run.

The Erlang/OTP standard distribution includes two testing frameworks: EUnit and Common Test. EUnit is mainly for unit testing and focuses on making it as simple as possible to write and run tests during development. Common Test is based on the so-called OTP Test Server and is a more heavy-duty framework that can run tests on one or several machines while the results are being logged to the machine that is running the framework; it's something you might use for large-scale testing like nightly integration tests. You can find more details about both these frameworks in the Tools section of the Erlang/OTP documentation.

We show briefly here what you need to do to write a test case using EUnit, because it's so simple. First, put this line of code in your source code, just after the `-module(...)` declaration:

```
-include_lib("eunit/include/eunit.hrl").
```

That was the hard part. Next, think of something to test; for example, you can test that you can successfully start the server. You must put the test in a function, which takes no arguments and whose name must end with `_test`. EUnit detects all such functions and assumes they're tests. A test succeeds if it returns some value and fails if it throws an exception. Hence, your test can be written

```
start_test() ->
    {ok, _} = tr_server:start_link(1055).
```

Recall that `=` is the match operator, which throws a `badmatch` error if the value of the right side doesn't match the pattern on the left. This means the only way this function can return normally is if the start operation succeeds; in every other case, `start_test()` results in an exception. Simple as that!

To run this test, you have to recompile the module. Then, from the Erlang shell, you can say either

```
eunit:test(tr_server).
```

or

```
tr_server:test().
```

This has the same effect: it runs all the tests in the `tr_server` module. Note that you never wrote a function called `test()`: EUnit creates this automatically, and it also ensures that all your test functions are exported.

Many more features in EUnit help you write tests as compactly as possible, including a set of useful macros that you get automatically when you include the eunit.hrl header file as you did earlier. We suggest that you read the EUnit Users Guide in the Erlang/OTP documentation for more information.

3.5 Summary

We've covered a lot of material in this chapter, going through all the basics of OTP behaviours and the three parts that make them what they are: the interface, the container, and the callback module. We've specifically covered the `gen_server` behaviour at some depth, through a real-world example.

In the next chapter, you'll hook this little stand-alone generic RPC server into a larger structure that will make it an enterprise-grade OTP application. When that's complete, your server will be part of an application that is versioned, fault tolerant, ready for use by others in their projects, and ready to go to production. This will add another layer to your basic understanding of Erlang/OTP as a framework by teaching you the fundamental structure for fault tolerance (the supervisor) and by teaching you how to roll up your functionality into nice OTP packages.

FUNCTIONAL PROGRAMMING

Erlang AND OTP IN ACTION

Martin Logan • Eric Merritt • Richard Carlsson

Free eBook
SEE INSERT

Erlang is an adaptable and fault tolerant functional programming language originally designed for the unique demands of the telecom industry. With Erlang/OTP's interpreter, compiler, database server, and libraries, developers are finding they can satisfy tough uptime and performance requirements in all kinds of other industries.

Erlang and OTP in Action teaches you the concepts of concurrent programming and the use of Erlang's message-passing model. It walks you through progressively more interesting examples, building systems in Erlang and integrating them with C/C++, Java, and .NET applications, including SOA and web architectures. This book is written for readers new to Erlang and interested in creating practical applications.

Build apps that...

- Never deadlock on a shared resource
- Keep running, even during code upgrades
- Recover gracefully from errors
- Scale unchanged from one to many processors
- Handle many simultaneous connections, and
- Maintain fast response times

A core developer for Erlware, **Martin Logan** has worked with Erlang since 1999. **Eric Merritt** is a core developer for Erlware and the Sinan build system. An Erlang pioneer, **Richard Carlsson** is an original member of the High-Performance Erlang group.

For online access to the authors and a free ebook for owners of this book, go to manning.com/ErlangandOTPinAction

"An enormous amount of experience, combined."
—From the Foreword by Ulf Wiger, Erlang Solutions Ltd.

"Full of practical, real-world code samples."
—Greg Donald
Vanderbilt University

"Illuminates how to do things the Erlang way."
—John S. Griffin
Overstock.com, Inc.

"The missing link on the Erlang learning curve."
—Ken Pratt, Ruboss Technology Corporation

"An indispensable resource."
—David Dossot
Programmer and Author

ISBN 13: 978-1-933988-78-8
ISBN 10: 1-933988-78-9
54999
9 781933 988788

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]