Kent Academic Repository

Full text document (pdf)

Citation for published version

Cesarini, Francesco and Thompson, Simon (2010) Erlang Behaviours: Programming With Process Design Patterns. In: Central European Functional Programming School, CEFP 2009.

DOI

Link to record in KAR

http://kar.kent.ac.uk/30616/

Document Version

UNSPECIFIED

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version.
Users are advised to check **http://kar.kent.ac.uk** for the status of the paper. **Users should always cite the published version of record.**

Enquiries
For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at **http://kar.kent.ac.uk/contact.html**

KAR Kent Academic Repository

University of Kent

Erlang Behaviours: Programming With Process Design Patterns

Francesco Cesarini
Erlang Solutions Ltd.,London, United Kingdom
www.erlang-solutions.com
email: francesco@erlang-solutions.com

Simon Thompson
School of Computing, University of Kent, Canterbury, United Kingdom
www.cs.kent.ac.uk/~sjt/
email: s.j.thompson@kent.ac.uk

Abstract: Erlang processes run independently of each other, each using separate memory and communicating with each other by message passing. These processes, while executing different code, do so following a number of common patterns. By examining different examples of Erlang-style concurrency in client/server architectures, we identify the generic and specific parts of the code and extract the generic code to form a process skeleton. In Erlang, the most commonly used patterns have been implemented in library modules, commonly referred to as OTP behaviours. They contain the generic code framework for concurrency and error handling, simplifying the complexity of concurrent programming and protecting the developer from many common pitfalls.

Keywords: Erlang, OTP, behaviour, generic, client/server, process, message passing, design pattern, concurrency, fault-tolerance.

0 Introduction

Processes in Erlang systems run concurrently in separate memory, and communicate with each other by message passing. Processes can be used for a wealth of applications, including as gateways to databases, as handlers for protocol stacks, and to manage the logging of trace messages from other processes. Although these processes handle different requests, there will be similarities in how these requests are handled. We call these similarities design patterns.

In these lecture notes, we will look at the particular example of the client/server process design pattern, abstracting out generic principles from specific examples. An experienced Erlang programmer will recognize these patterns in the design phase of the project, and so will use libraries and templates that are part of the OTP framework. Section 1 gives a brief introduction to Erlang, providing the necessary background to the rest of the chapter. Section 2 of these lecture notes introduces the concept of an Erlang process skeleton, a pattern followed by most processes irrespective of their behaviour or function. Section 3 introduces client/server behaviours in Erlang processes, using an example taken from mobile telephony. Section 4 takes this example, and re-implements it using the `gen_server` OTP behaviour library. These lecture notes are based on the authors' book *Erlang Programming*, ISBN: 978-0-596-51818-9 published in 2009 by O'Reilly Media.

1 Erlang

This section gives a brief overview of those aspects of Erlang covered in these notes; more details of these and other aspects of Erlang and the OTP library can be found in the online documentation for the language as well as in our book.

Erlang is at basis a functional language, with no side-effects due to assignment since Erlang contains single assignment: each (instance of a) variable can only be assigned to once, so that variables assignments play the role of definitions in other languages. An example module is given now

```
-module(factorial).
-export([fac/1]).
```

```
fac(0) -> 1;
fac(N) when N>0 ->
       Prev = fac(N-1),
       n*Prev.
```

This contains an assignment to `Prev`, as well as a simple case of definition by pattern matching. The clauses of the function definition are separated by semi-colons, and the first head matching the argument is used. In this example, the first clause gives the factorial of zero, the second factorials of positive numbers. The body of each clause is a sequence of expressions, and the result of that clause is the final expression in the body.

Within the module functions are called in the usual way; outside, the name of the module is prepended as in `factorial:fac(3)`. It is possible to define functions with the same name but different numbers of arguments – this is called their "arity". In the export directive in the `factorial` module the `fac` function of arity one is denoted by `fac/1`.

Erlang contains tuples (or product types) and lists. Tuples are enclosed in curly brackets, as in `{ok,37}`; lists in square brackets `[23,34]`. The notation `[X|Xs]` matches a non empty list with head `X` and tail `Xs`. Identifiers beginning with a lower case letter denote atoms, which simply stand for themselves; the '`ok`' in the tuple `{ok,37}` is an example of an atom. Atoms used in this way are often used to distinguish between different kinds of function result: as well as '`ok`' results, there might be results of the form `{error, "Error string"}`.

Erlang concurrency is by message passing between processes, each executing in a separate memory space. Processes are identified by process identifiers, called 'Pid's, but processes can also be registered under a name; this should only be used for long-lived, "static" processes. A message `Msg` is sent to a process with process id `Pid` thus: `Pid ! Msg`. A process can find out its pid by calling the built-in function (BIF) `self/0`, and this can then be sent to other processes for them to use to communicate with the original process.

Suppose that a process expects to receive messages of the form `{ok, N}` and `{error, St}`. To process these it uses a `receive` statement

```
receive
       {ok, N} ->
              N+1;
       {error, _} ->
              0
end
```

The result of this is a number, with the particular result determined by pattern matching. When the value of a variable is not needed, the wild-card '_' can be used, as shown.

Message passing between processes is asynchronous, and the messages received by a process are placed in the process's mailbox in the order in which they arrive. Suppose that now the `receive` statement above is to be executed: if the first element in the mailbox is either `{ok, N}` or `{error, St}`: the corresponding result will be returned. If the first message in the mailbox is not of this form, it is retained in the mailbox, and the second is processed in a similar way. If no messages match, then the `receive` will wait for a matching message to be received.

The remainder of these notes give many examples of concurrent Erlang processes, and we move to looking at those now.

2 Process Skeletons

There is a common pattern to the behaviour of processes, regardless of the particular purpose for which the process was created. To start with, a processes has to be spawned and then, optionally, have its alias registered. The first action of the newly spawned process is to initialize the process loop data. The loop data is often the result of arguments passed to the spawn built-in function (BIF) at the initialization of the process. It loop data is stored in a variable we refer to as the process state. The state is passed to a

receive-evaluate function, running a loop which receives a message, handles it, updates the state, and passes it back as an argument to a tail-recursive call. If one of the messages it handles is a 'stop' message, the receiving process will clean up after itself and then terminate.

This is a recurring design among processes that we usually refer to as a design pattern, and it will occur regardless of the task the process has been assigned to perform in the body of the loop. Figure 1 shows an example skeleton.

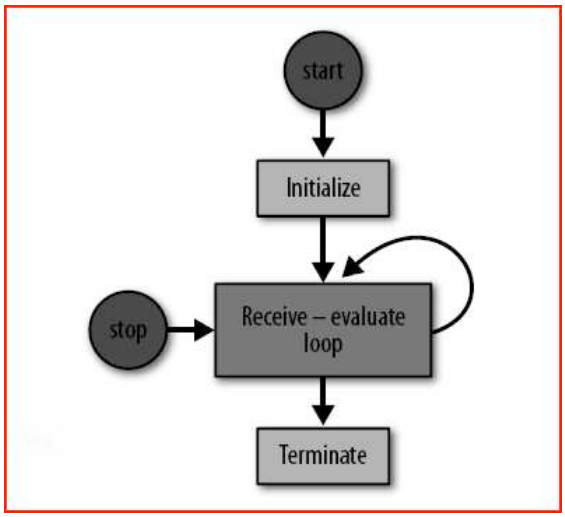

Fig. 1. Process skeleton.

Let's now look at the differences between the particular processes which conform to this pattern:

- The arguments passed to the spawn BIF calls will differ from one process to another.
- You have to decide whether you should register a process under an alias, and, if you do register it, what alias should be used.
- In the function that initializes the process state, the actions taken will differ based on the tasks the process will perform.
- The state of the system is represented by the loop data in every case, but the contents of the loop data will vary among processes.
- When in the body of the receive-evaluate loop, processes will receive different messages and handle them in different ways.
- Finally, on termination, the cleanup will vary from process to process.

So, even if a skeleton of generic actions exists, these actions are complemented by specific ones that are directly related to the specific tasks assigned to the process. Using this skeleton as a template, one can create processes which act as servers, as finite state machines, as event handlers and as supervisors. In the following sections, we will concentrate on client/server models.

3 Client/Server Models

Erlang processes can be used to implement client/server solutions, where both clients and servers are represented as Erlang processes. A server could be a FIFO queue to a printer, a window manager, or a file server. The resources it handles could be a database, calendar, or finite list of items such as rooms, books, or radio frequencies. Clients access these resources by sending the server a request to print a file, to update a window, to book a room, or to use a frequency. The server receives the request, handles it, and responds with an acknowledgment and a return value if the request was successful, or with an error if the request did not succeed.

When implementing client/server behaviour, clients and servers are represented as Erlang processes. Interaction between them takes place through the sending and receiving of messages. Message passing is often hidden in functional interfaces, so that instead of calling:

```
printerserver ! {print, File}
```

a client would call a `print` function, as in:

```
printerserver:print(File)
```

This is a form of information hiding, where we do not make the client aware that the server is a process, that it could be registered, and that it might reside on a remote computer. Nor do we expose the message protocol being used between the client and the server, keeping the interface between them safe and simple. All the client needs to do is call a function and use with the return value of the fucntion.

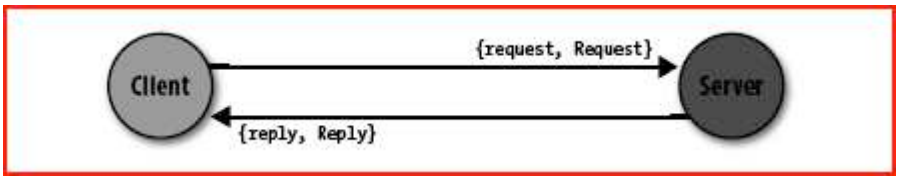

Fig. 2. Synchronous message passing

If a client using the service or resource handled by the server expects a reply to the request, the implementation of the call to the server has to be synchronous, as in Figure 2. If the client does not need a reply, the call to the server can be asynchronous. When you encapsulate synchronous and asynchronous calls in a function call, the function commonly returns the atom `ok`, indicating that the request was sent to the server. Functions encapsulating synchronous calls will return the value expected by the client. These return values usually follow the format `ok`, `{ok, Result}`, when the result is successful or `{error, Reason}` when it is unsuccessful. In the latter case the `Reason` encapsulates why the request has failed.

3.1 A Client/Server Example

So that you understand what we are talking about, let's walk through a client/server example and test it in the shell. This server is responsible for managing radio frequencies on behalf of its clients, the mobile phones connected to the network. The phone requests a frequency whenever a call needs to be connected, and releases it once the call has terminated (see Fig. 3 below).

When a mobile phone has to set up a connection to another subscriber, it calls the client function `frequency:allocate/0`. This call has the effect of generating a synchronous message which is sent to the server. The server handles it and responds with either a message containing an available frequency or an error if all frequencies are being used. The result of the `allocate/0` call will therefore be either `{ok, Frequency}` or `{error, no_frequencies}`.

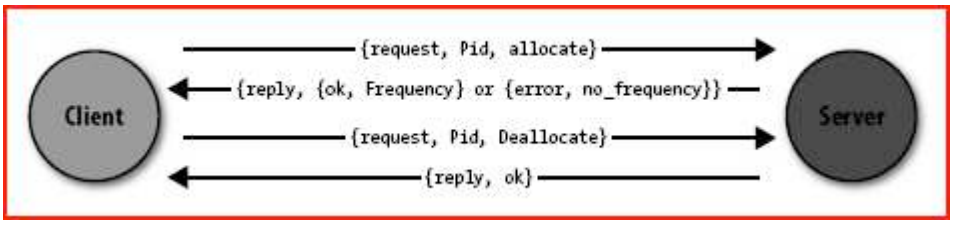

Fig. 3. Frequency server message sequence diagram

Through a functional interface, we hide the message-passing mechanism, the format of these messages, and the fact that the frequency server is implemented as a registered Erlang process. If we were to move the server to a remote host, we could do so without having to change the client interface.

When the client has completed its phone call and releases the connection, it needs to deallocate the frequency so that other clients can reuse it. It does so by calling the client function `frequency:deallocate(Frequency)`. The call results in a message being sent to the server. The server can then make the frequency available to other clients and responds with the atom `ok`. The atom is sent back to the client and becomes the return value of the `deallocate/1` call. Figure 3 above shows the message sequence diagram of this example.

The code for the server is in the `frequency` module. Here is the first part:

```
-module(frequency).
-export([start/0, stop/0, allocate/0, deallocate/1]).
-export([init/0]).

%% These are the start functions used to create and
%% initialize the server.

start() ->
    register(frequency, spawn(frequency, init, [])).

init() ->
    Frequencies = {get_frequencies(), []},
    loop(Frequencies).
```

When spawning a process, you have to export the `init/0` function because it is used by the `spawn/3` BIF. We have put this function in a separate `export` clause to distinguish it from the client functions, which are supposed to be called from other modules. On the other hand, calling `frequency:init()` explicitly anywhere in your code is considered to be very bad practice.

The newly spawned process starts executing in the `init/0` function. It creates a tuple consisting of the available frequencies, retrieved through the `get_frequencies/0` call, and a list of the allocated frequencies—initially given by the empty list—as the server has just been started. The tuple, which forms the state or loop data, is bound to the `Frequencies` variable and passed as an argument to the receive-evaluate function, which in this example we've called `loop/1`.

In the `init/0` function, we use the variable `Frequencies` for readability reasons, but nothing is stopping us from creating the tuple directly in the call thus `loop({get_frequencies(), []})`. Here is how the client functions are implemented:

```
%% The client Functions
stop()           -> call(stop).
allocate()       -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).

%% We hide all message passing and the message
%% protocol in a functional interface.

call(Message) ->
    frequency ! {request, self(), Message},
    receive
        {reply, Reply} -> Reply
    end.
```

Client and supervisor processes can interact with the frequency server using what we refer to as client functions. These exported functions include `start`, `stop`, `allocate`, and `deallocate`. They call the `call/1` function, passing the message to be sent to the server as an argument. This function will encapsulate the message protocol between the server and its clients, sending a message of the format `{request, Pid, Message}`. The atom `request` is a tag in the tuple, `Pid` is the process identifier of the calling process (returned by calling the `self/0` BIF in the calling process), and `Message` is the argument originally passed to the `call/1` function.

When the message has been sent to the process, the client is suspended in the `receive` clause waiting for a response of the format `{reply, Reply}`, where the atom `reply` is a tag and the variable `Reply` is the actual response. The server response is pattern-matched, and the contents of the variable `Reply` become the return value of the client functions.

Pay special attention to how message passing and the message protocol have been abstracted to a format independent of the action relating to the message itself; this is a form of information hiding, that allows the details of the protocol and the message structure to be modified without affecting any of the client code. Now that we have covered the code to start and interact with the frequency server, let's take a look at its receive-evaluate loop:

```
%% The Main Loop
loop(Frequencies) ->
    receive
        {request, Pid, allocate} ->
            {NewFrequencies, Reply} = allocate(Frequencies, Pid),
             reply(Pid, Reply),
             loop(NewFrequencies);
        {request, Pid , {deallocate, Freq}} ->
             NewFrequencies = deallocate(Frequencies, Freq),
             reply(Pid, ok),
             loop(NewFrequencies);
        {request, Pid, stop} ->
             reply(Pid, ok)
    end.

reply(Pid, Reply) ->
    Pid ! {reply, Reply}.
```

The `receive` clause will accept three kinds of requests originating from the client functions, namely `allocate`, `deallocate`, and `stop`. These requests follow the format defined in the `call/1` function, that is, `{request, Pid, Message}`. The `Message` is pattern-matched in the expression and used to determine which clause is executed. This, in turn, determines the internal functions that are called. These internal functions will return the new loop data, which in our example consists of the pair of lists of available and allocated frequencies, and where needed, a reply to send back to the client. The client `Pid`, sent as part of the request, is used to identify the calling process and is used in the `reply/2` call.

Assume a client wants to initiate a call. To do so, it would request a frequency by calling the `frequency:allocate/0` function. This function sends a message of the format `{request, Pid, allocate}` to the frequency server, pattern matching in the first clause of the `receive` statement. This message will result in the server function `allocate(Frequencies, Pid)` being called, where `Frequencies` is the loop data containing a tuple of allocated and available frequencies. The `allocate` function will check whether there are any available frequencies:

```
allocate({[], Allocated}, _Pid) ->
    {{[], Allocated}, {error, no_frequencies}};
allocate({[Freq|Frequencies], Allocated}, Pid) ->
    link(Pid),
    {{Frequencies,[{Freq,Pid}|Allocated]},{ok,Freq}}.
```

If there are frequencies available, it will return the updated loop data, where the newly allocated frequency has been moved from the available list and stored together with the `Pid` in the list of allocated frequencies. The reply sent to the client is of the format `{ok, Frequency}`.

If no frequencies are available, the loop data is unchanged and the `{error, no_frequency}` message is returned as a reply.

After calling the `allocate` function, the reply is sent to the client by calling `reply(Pid, Message)`, which formats the message according to the internal client/server message format and sends it back to the client. Finally, the function `loop/1` is called recursively, passing the new loop data as an argument.

Deallocation works in a similar way. The client function call results in the message `{request, Pid, {deallocate, Frequency}}` being sent and matched in the second clause of the `receive` statement. This makes a call to `deallocate(Frequencies, Frequency)` and the `deallocate` function moves the Frequency from the allocated list to the deallocated one, returning the updated loop data. The atom `ok` is sent back to the client, and the `loop/1` function is called recursively with the updated loop data.

If the `stop` request is received, `ok` is returned to the calling process and the server terminates, as there is no more code to execute. In the previous two clauses, `loop/1` was called in the final expression of the `receive` clause, but not in this case.

We complete this system by implementing the deallocation function, which assumes that it is only called when the frequency to be deallocated is indeed allocated:

```
deallocate({Free, Allocated}, Freq) ->
   NewAllocated=lists:keydelete(Freq, 1, Allocated),
  {[Freq|Free], NewAllocated}.
```

The `allocate/2` and `deallocate/2` functions are local to the `frequency` module, and are what we refer to as internal help functions. You can see an example of the frequency allocator in action now:

```
1> c(frequency).
{ok,frequency}
2> frequency:start().
true
3> frequency:allocate().
{ok,10}
4> frequency:allocate().
{ok,11}
5> frequency:allocate().
{ok,12}
6> frequency:allocate().
{ok,13}
7> frequency:allocate().
{ok,14}
8> frequency:allocate().
{ok,15}
9> frequency:allocate().
{error,no_frequency}
10> frequency:deallocate(11).
ok
11> frequency:allocate().
{ok,11}
12> frequency:stop().
ok
```

3.2 A Process Pattern Example

In this section we look at two other client-server examples, and when doing so, we compare and contrast them to the frequency server we described in the previous section. Picture an application, either a web browser or a word processor, which handles many simultaneously open windows centrally controlled by a window manager. As we aim to have a process for each truly concurrent activity, spawning a process for every window is the way to go. These processes would probably not be

registered, as many windows of the same type could be running concurrently, so communication to them is by means of their Pid.

After being spawned, each process would call the initialize function, which draws and displays the window and its contents. The return value of the initialize function contains references to the widgets displayed in the window. These references are stored in the state variable and are used whenever the window needs updating. The state variable is passed as an argument to a tail-recursive function that implements the receive-evaluate loop.

In this loop function, the process waits for events originating in or relating to the window it is managing. It could be a user typing in a form or choosing a menu entry, or an external process pushing data that needs to be displayed. Every event relating to this window is translated to an Erlang message and sent to the process. The process, upon receiving the message, calls the handle function, passing the message and state as arguments. If the event were the result of a few keystrokes typed in a form, the handle function might want to display them. If the user picked an entry in one of the menus, the handle function would take appropriate actions in executing that menu choice. Or, if the event was caused by an external process pushing data, possibly an image from a webcam or an alert message, the appropriate widget would be updated. The receipt of these events in Erlang would be seen as a generic pattern in all processes. What would be considered specific and change from process to process is how these events are handled.

Finally, what if the process receives a stop message? This message might have originated from a user picking the Exit menu entry or clicking the Destroy button, or from the window manager broadcasting a notification that the application is being shut down. Regardless of the reason, a stop message is sent to the process. Upon receiving it, the process calls a terminate function, which destroys all of the widgets, ensuring that they are no longer displayed. After the window has been shut down, the process terminates because there is no more code to execute.

Look at the following process skeleton. Could you not fit all of the specific code into the `initialize/1`, `handle_msg/2`, and `terminate/1` functions for not only the window example, but also the frequency server?

```
-module(server).
-export([start/2, stop/1, call/2]).
-export([init/1]).

start(Name, Data) ->
    Pid = spawn(server, init,[Data]),
    register(Name, Pid),
    ok.

stop(Name) ->
    Name ! {stop, self()},
    receive {reply, Reply} -> Reply end.

call(Name, Msg) ->
    Name ! {request, self(), Msg},
    receive {reply, Reply} -> Reply end.

reply(To, Msg) ->
    To ! {reply, Msg}.

init(Data) ->
    loop(initialize(Data)).

loop(State) ->
    receive
       {request, From, Msg} ->
           {Reply,NewState} = handle_msg(Msg, State),
           reply(From, Reply),
           loop(NewState);
```

```
        {stop, From} ->
            reply(From, terminate(State))
    end.

initialize(...)      -> ...
handle_msg(...,...) -> ...
terminate(...)       -> ...
```

Using the generic code in the preceding skeleton, let's go through the GUI example one last time:

- The `initialize/1` function draws the window and displays it, returning a reference to the widget that gets bound to the state variable.
- Every time an event arrives in the form of an Erlang message, the event is taken care of in the `handle_msg` function. The call takes the message and the state as arguments and returns an updated `State` variable. This variable is passed to the recursive loop call, ensuring that the process is kept alive. Any reply is also sent back to the process where the request originated.
- If the `stop` message is received, `terminate/1` is called, destroying the window and all the widgets associated with it. The `loop` function is not called, allowing the process to terminate normally.

This server skeleton example actually exists for client/servers, finite state machines, event handlers and supervisor processes as library modules which come as part as the OTP middleware. In the next section, we describe the client–server behaviour, often referred to as the `gen_server`.

4 OTP Behaviours

In previous section, we introduced patterns that recur when you program using the Erlang concurrency model. We discussed functionality common to concurrent systems, and you saw that processes will handle very different tasks in a similar way. We also emphasized special cases and potential problems that have to be handled when dealing with concurrency.

Picture a project with 50 developers spread across several geographic locations. If the project is not properly coordinated and no templates are provided, how many different client/server implementations might the project end up with? Even more dangerous, how many of these implementations will handle special borderline cases and concurrency-related errors correctly, if at all? Without a code review, can you be sure there is a uniform way across the system to handle server crashes that occur after clients have sent a request to the server? Or guarantee that the response from a request is indeed the response, and not just any message that conforms to the internal message protocol?

OTP behaviours address all of these issues by providing library modules that implement the most common concurrent design patterns. Behind the scenes, without the programmer having to be aware of it, the library modules ensure that errors and special cases are handled in a consistent way. As a result, OTP behaviours provide a set of standardized building blocks used in designing and building industrial-grade systems. The subject of OTP behaviours and their related middleware is vast. In this section, we provide the overview you need to get started.

4.1 Introduction

OTP behaviours are a formalization of process design patterns. They are implemented in library modules that are provided with the standard Erlang distribution. These library modules do all of the generic process work and error handling. The specific code, written by the programmer, is placed in a separate module and called through a set of predefined callback functions.

OTP behaviours include worker processes, which do the actual processing, and supervisors, whose task is to monitor workers and other supervisors. Worker behaviours, often denoted in diagrams as circles, include servers, event handlers, and finite state machines. Supervisors, denoted in illustrations as squares, monitor their children, both workers and other supervisors, creating what is called a supervision tree

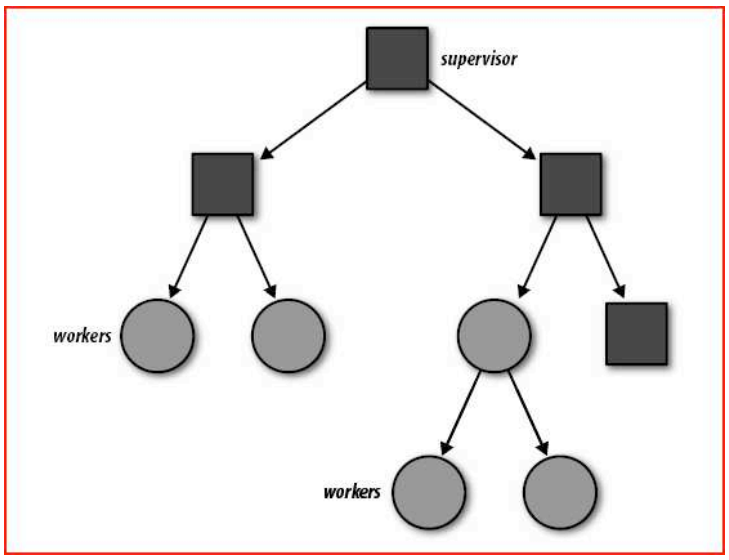

Fig. 4. OTP Supervision Tree

Supervision trees are packaged into a behaviour called an application. OTP applications not only are the building blocks of Erlang systems, but also are a way to package reusable components. Industrial-grade systems consist of a set of loosely-coupled, possibly distributed applications. These applications are part of the standard Erlang distribution or are specific applications developed by you, the programmer.

Do not confuse OTP applications with the more general concept of an application, which usually refers to a more complete system that solves a high-level task. Examples of OTP applications include the Mnesia database or the Simple Network Management Protocol (SNMP) agent. An OTP application is a reusable component that packages library modules together with supervisor and worker processes. From now on, when we refer to an application, we will mean an OTP application.

The behaviour module contains all of the generic code. Although it is possible to implement your own behaviour module, doing so is rare because the behaviour modules that come as part of the Erlang/OTP distribution will cater to most of the design patterns you would use in your code. The generic functionality provided in a behaviour module includes operations such as the following:

- Spawning and possibly registering the process
- Sending and receiving client messages as synchronous or asynchronous calls, including defining the internal message protocol
- Storing the loop data and managing the process loop
- Stopping the process

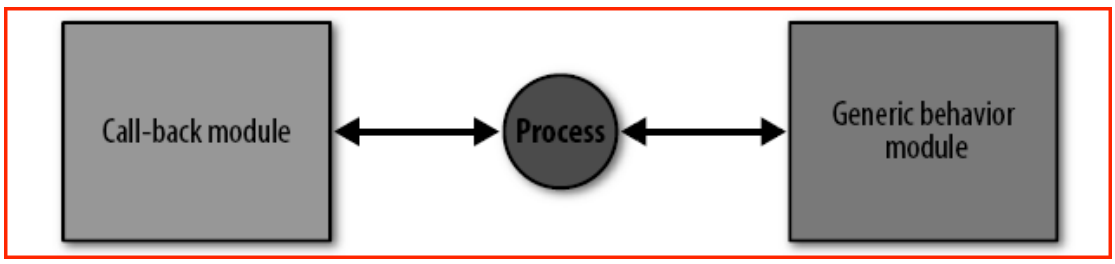

Fig. 5. Splitting the code in generic and specific modules

Although the behaviour module is provided, the programmer has to develop the callback module. A callback module contains all of the specific code required to deliver the desired functionality. The specific code is invoked through a callback interface that is standardized for each behaviour.

The loop data is a variable that will contain the data the behaviour needs to store in between calls. After the call, an updated variant of the loop data is returned. This updated loop data, often referred to as the new loop data, is passed as an argument in the next call. Loop data is also often referred to as the behaviour state.

The functionality to be included in the callback module for the generic server application to deliver the specific required behaviour includes the following:

- Initializing the process loop data, and, if the process is registered, the process name.
- Handling the specific client requests, and, if synchronous, the replies sent back to the client.
- Handling and updating the process loop data in between the process requests.
- Cleaning up the process loop data upon termination.

There are many advantages to splitting the code into generic behaviour libraries and specific callback modules:

- Because many of the special cases and errors that might occur are already handled in the solid, well-tested behaviour library, you can expect fewer bugs in your product.
- For this reason, and also because so much of the code is already written for you, you can expect to have a shorter time to market.
- It forces the programmer to write code in a way that avoids errors typically found in concurrent applications.
- Finally, your whole team will come to share a common programming style.

When reading someone else's code while armed with a basic comprehension of the existing behaviours, no effort is required to understand the client/server protocol, looking for where and how processes are started or terminated, or how the loop data is handled. All of it is managed by the generic behaviour library. Instead of having to focus on how everything is done, you can focus on what is being done specifically in this case, as coded in the callback module.

4.2 Generic Servers

Generic servers that implement client/server behaviours are defined in the `gen_server` behaviour that comes as part of the standard library application. In explaining generic servers, we will use the frequency server example from the client server section.

We will rewrite the `frequency.erl` module, migrating it from an Erlang process to a `gen_server` behaviour. In doing so, we will not touch the client interface, keeping the API as it is. When working your way through the example, if you are interested in the details, have the online Erlang manual pages for the `gen_server` module to hand.

4.3 Starting Your Server

With the `gen_server` behaviour, instead of using the `spawn` and `spawn_link` BIFs, you will use the `gen_server:start/4` and `gen_server:start_link/4` functions.

The main difference between `spawn` and `start` is the synchronous nature of the call. Using start instead of spawn makes starting the worker process more deterministic and prevents unforeseen race conditions, as the call will not return the pid of the worker until it has been initialized. You call the functions as follows (we show two variants for each of the functions):

```
gen_server:start_link(ServerName, CallBackModule, Arguments, Options)
gen_server:start(ServerName, CallBackModule, Arguments, Options)

gen_server:start_link(CallBackModule, Arguments, Options)
gen_server:start(CallBackModule, Arguments, Options)
```

In the preceding calls:

- `ServerName` is a tuple of the format `{local, Name}` or `{global, Name}`, denoting a local or global `Name` for the process if it is to be registered. If you do not want to register the process and instead reference it using its pid, you omit the argument and use a `start_link/3` or `start/3` function call instead.

- `CallbackModule` is the name of the module in which the specific callback functions are placed.

- `Arguments` is a valid Erlang term that is passed to the `init/1` callback function. You can choose what type of term to pass: if you have many arguments to pass, use a list or a tuple; if you have none, pass an atom or an empty list, ignoring it in the callback function.

- `Options` is a list that allows you to set the memory management flags `fullsweep_after` and `heapsize`, as well as tracing and debugging flags. Most behaviour implementations just pass the empty list.

The `start` functions will spawn a new process that calls the `init(Arguments)` callback function in the `CallbackModule`, with the `Arguments` supplied. The `init` function must initialize the `LoopData` of the server and has to return a tuple of the format `{ok, LoopData}`. `LoopData` contains the first instance of the loop data that will be passed between the callback functions. If you want to store some of the arguments you passed to the init function, you would do so in the `LoopData` variable.

The obvious difference between the `start_link` and `start` functions is that `start_link` links to its parent and `start` doesn't. This needs a special mention, however, as it is an OTP behaviour's responsibility to link itself to the supervisor. The `start` functions are often used when testing behaviours from the shell, as a typing error causing the shell process to crash would not affect the behaviour. All variants of the `start` and `start_link` functions return `{ok, Pid}`.

Before going ahead with the example, let's quickly review what we have discussed so far. You start a `gen_server` behaviour using the `gen_server:start_link` call. This results in a new process that calls the `init/1` callback function. This function initializes the `LoopData` and returns the tuple `{ok, LoopData}`.

In our example, we call `start_link/4`, registering the process with the same name as the callback module, using the `?MODULE` macro call. We don't pass any arguments, and as a result, just send the empty list. The options list is kept empty:

```
start() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

init(_Args) ->
    {ok, {get_frequencies(), []}}.

get_frequencies() -> [10,11,12,13,14,15].
```

Although the supervisor process might call the `start_link/4` function, the `init/1` call-back is called by a different process: the one that was just spawned. Our `LoopData` contains the tuple of available and allocated frequencies. If the server `LoopData` does not need to be passed inbetween calls, a value still has to be included when returning the `{ok, LoopData}` structure. We get around it by returning the atom `null`.

Do only what is necessary and minimize the operations in your `init` function, as the call to `init` is a synchronous call that prevents all of the other serialized processes from starting until it returns.

4.4 Passing Messages

If you want to send a message to your server, you use the following calls:

```
gen_server:cast(Name, Message)
gen_server:call(Name, Message)
```

In the preceding calls:

- `Name` is either the local registered name of the server or the tuple `{global, Name}`. It could also be the process identifier of the server.

- `Message` is a valid Erlang term containing a message passed on to the server. For asynchronous message requests, you use `cast/2`. If you're using a pid, the call will immediately return the atom `ok`, regardless of whether the `gen_server` to which you are sending the message is alive. These semantics are no different from the standard `Name ! Message` construct, where if the registered process `Name` does not exist, the calling process terminates.

Upon receiving the message, `gen_server` will call the function `handle_cast(Message, LoopData)` in the callback module. `Message` is the argument passed to the `cast/2` function, and `LoopData` is the argument originally returned by the `init/1` callback function. The `handle_cast/2` callback function handles the specifics of the message, and upon finishing, it has to return the tuple `{noreply, NewLoopData}`. In future calls to the server, the `NewLoopData` value most recently returned will be passed as an argument when a message is sent to the server.

If you want to send a synchronous message to the server, you use the `call/2` function. Upon receiving this message, the process uses the `handle_call(Message, From, LoopData)` function in the callback module. It contains specific code for the particular server, and having completed, it returns the tuple `{reply, Reply, NewLoopData}`. Only now does the `call/3` function synchronously return the value `Reply`. If the process to which you are sending a message does not exist, regardless of whether it is registered, the process invoking the call function terminates.

Let's start by taking two functions from our service API; we will provide the whole program later. They are called by the client process and result in a synchronous message being sent to the server process registered with the same name as the callback module. Note that validating the data sent to the server should occur on the client side. If the client sends incorrect information, the server should terminate.

```
allocate()        ->
    gen_server:call(?MODULE, {allocate, self()}).
deallocate(Freq) ->
    gen_server:call(?MODULE, {deallocate, Freq}).
```

Upon receiving the messages, the gen_server process calls the `handle_call/3` callback function dealing with the messages in the same order in which they were sent:

```
handle_call({allocate, Pid}, _From, Frequencies) ->
    {NewFrequencies, Reply} = allocate(Frequencies, Pid),
    {reply, Reply, NewFrequencies};
handle_call({deallocate, Freq}, _From, Frequencies) ->
    NewFrequencies=deallocate(Frequencies, Freq),
    {reply, ok, NewFrequencies}.
```

Note the return value of the callback function. The tuple contains the control atom `reply`, telling the `gen_server` generic code that the second element of the tuple is the `Reply` to be sent back to the client. The third element of the tuple is the new `LoopData`, which, in a new iteration of the server, is passed as the third argument to the `handle_call/3` function; in both cases here it is unchanged. The argument `_From` is a tuple containing a unique message reference and the client process identifier.

The tuple as a whole is used in library functions that we will not be discussing in this article. In the majority of cases, you will not need it.

The `gen_server` library module has a number of mechanisms and safeguards built in that function behind the scenes. If your client sends a synchronous message to your server and you do not get a response within five seconds, the process executing the `call/2` function is terminated. You can override this by using the following code:

```
gen_server:call(Name, Message, Timeout)
```

where `Timeout` is a value in milliseconds or the atom `infinity`. The timeout mechanism was originally put in place for deadlock prevention purposes, ensuring that servers that accidentally call each other are terminated after the default timeout. The crash report would be logged, and hopefully would result in a patch. Most applications will function appropriately with a timeout of five seconds, but under very heavy loads, you might have to fine-tune the value and possibly even use `infinity`; this choice is very application-dependent. All of the critical code in Erlang/OTP uses `infinity`.

Other safeguards when using the `gen_server:call/2` function include the case of sending a message to a non-existing server as well as the case that a server that crashes before sending its reply. In both cases, the calling process will terminate. In raw Erlang, sending a message that is never pattern-matched in a receive clause is a bug that can cause a memory leak.

What do you think happens if you do a `call` or a `cast` to your server, but do not handle the message in the `handle_call/3` and `handle_cast/2` calls, respectively? In OTP, when a `call` or a `cast` is called, the message will always be extracted from the process mailbox and the respective callback functions are invoked. If none of the callback functions pattern-matches the message passed as the first argument, the process will crash with a function clause error. As a result, such issues will be caught in the early stages of the testing phase and dealt with accordingly.

4.5 Stopping the Server

How do you stop the server? In your `handle_call/3` and `handle_cast/2` callback functions, instead of returning `{reply, Reply, NewLoopData}` or `{noreply, NewLoopData}`, you can return `{stop, Reason, Reply, NewLoopData}` or `{stop, Reason, NewLoopData}`, respectively. Something has to trigger this return value, often a stop message sent to the server. Upon receiving the stop tuple containing the `Reason` and `LoopData`, the generic code executes the `terminate(Reason, LoopData)` callback.

The `terminate` function is the natural place to insert the code needed to clean up the `LoopData` of the server and any other persistent data used by the system. The `stop` call does not have to occur within a synchronous call, so let's use `cast` when implementing it:

```
stop() ->
    gen_server:cast(?MODULE, stop).

handle_cast(stop, Frequencies) ->
    {stop, normal, Frequencies}.

terminate(_Reason, _Frequencies) ->
    ok.
```

Remember that `stop/0` will be called by the client process, while the `handle_cast/2` and `handle_call/2` functions are called by the behaviour process. In the `handle_cast/2` callback, we return the reason normal in the `stop` construct. Any reason other than normal will result in an error report being generated.

With thousands of generic servers potentially being spawned and terminated every second, generating error reports for every one of them is not the way to go. You should return a non-normal value only if

something that should not have happened occurs and you have no way to recover. A socket being closed or a corrupt message from an external source should not promot a non-normal exit reason.

Use of the behaviour callbacks as library functions and invoking them from other parts of your program is an extremely bad practice. For example, you should never call `frequency:init(FileName)` from another module to retrieve the initial loop data. Calls to behaviour callback functions should originate only from the behaviour library modules as a result of an event occurring in the system, and never directly by the user.

The Example in Full
Here is the `frequency.erl` module in full, rewritten as a `gen_server` behaviour:

```
% File: frequency.erl
%% Purpose gen_server call back module for the frequency
%% allocator

-module(frequency2).
-export([start/0, stop/0, allocate/0, deallocate/1]).
-export([init/1, terminate/2, handle_cast/2, handle_call/3]).

%% The start and stop Functions
start() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

stop() ->
    gen_server:cast(?MODULE, stop).

%%  The client Functions
allocate()       ->
    gen_server:call(?MODULE, {allocate, self()}).
deallocate(Freq) ->
    gen_server:call(?MODULE, {deallocate, Freq}).

%% Callback functions
handle_call({allocate, Pid}, _From, Frequencies) ->
    {NewFrequencies, Reply} = allocate(Frequencies, Pid),
    {reply, Reply, NewFrequencies};

handle_call({deallocate, Freq}, _From, Frequencies) ->
    NewFrequencies=deallocate(Frequencies, Freq),
    {reply, ok, NewFrequencies}.

handle_cast(stop, Frequencies) ->
    {stop, normal, Frequencies}.

init(_Args) ->
    {ok, {get_frequencies(), []}}.

terminate(_Reason, _Frequencies) ->
    ok.

%% Local Functions
get_frequencies() -> [10,11,12,13,14,15].

allocate({[], Allocated}, _Pid) ->
    {{[], Allocated}, {error, no_frequencies}};
allocate({[Freq|Frequencies], Allocated}, Pid) ->
    {{Frequencies,[{Freq,Pid}|Allocated]},{ok,Freq}}.

deallocate({Free, Allocated}, Freq) ->
    {value,{Freq, _Pid}}= lists:keysearch(Freq,1,Allocated),
```

```
    NewAllocated=lists:keydelete(Freq,1,Allocated),
    {[Freq|Free],  NewAllocated}.
```

Running the `gen_server`
When testing the `gen_server` instance in the shell, you get exactly the same behaviour as when you used the server process that you coded yourself. However, the code is more solid, as deadlocks, server crashes, timeouts, and other errors related to concurrent programming are handled behind the scenes. The following calls:

```
start(Name, Mod, Arguments, Opts)
start_link(Name, Mod, Arguments, Opts),
```

where `Name` is an optional argument, spawn a new process. The process will result in the callback function `init(Arguments)` being called, which should return one of the values `{ok, LoopData}` or `{stop, Reason}`. If `init/1` returns `{stop, Reason}` the `terminate/2` "cleanup" function will not be called.

Synchronous communication
Use `call(Name, Msg)` to send a synchronous message to your server. It will result in the callback function `handle_call(Msg, From, LoopData)` being called by the server process. The expected return values include `{reply, Reply, NewLoopData}` and `{stop, Reason, Reply, NewLoopData}`.

Asynchronous communication
If you want to send an asynchronous message, use `cast(Name, Msg)`. It will be handled in the `handle_cast(Msg, LoopData)` callback function, returning either `{noreply, NewLoopData}` or `{stop, Reason, NewLoopData}`.

Non-OTP-compliant messages
Upon receiving non-OTP-compliant messages, `gen_server` will execute the `handle_info(Msg, LoopData)` callback function. The function should return either `{noreply, NewLoopData}` or `{stop, Reason, NewLoopData}`.

Termination
Upon receiving a `stop` construct from one of the callback functions (except for `init`), the `terminate(Reason, LoopData)` callback is invoked. In `terminate/2`, you would typ-ically undo things you did in `init/1`. Its return value is ignored.

Other Behaviours

Finite state machines are a crucial component of telecom systems. The `gen_fsm` module provides you with a behaviour that you can use to implement processes acting as finite state machines. States are defined as callback functions that return a tuple containing the next State and the updated loop data. You can send events to these states synchronously and asynchronously. The finite state machine callback module should also export the standard callback functions such as `init`, `terminate`, and `handle_info`. Examples of processes acting as finite state machines include protocol stacks, communication layers, mutex semaphores as well as high level control flow in telephony systems.

Event handlers and managers are another behaviour implemented in the `gen_event` library module. The idea is to create a centralized point that receives events of a specific kind. Events can be sent synchronously and asynchronously with a predefined set of actions being applied when they are received. Possible responses to events include logging them to file, sending off an alarm in the form of an SMS, or collecting statistics. Each of these actions is defined in a separate callback module with its own loop data, preserved in between calls. Handlers can be added, removed, or updated for every specific event manager. So, in practice, for every event manager, there could be many callback

modules, and different instances of these callback modules could exist in different managers. Event handlers include processes receiving alarms, live trace data, equipment related events or simple logs.

The `supervisor` behaviour's task is to monitor its children and, based on some preconfigured rules, take action when they terminate. The children that make up the supervision tree include both supervisors and worker processes. Worker processes are OTP behaviours including `gen_server` and `gen_event`.

Worker processes have to link themselves to the `supervisor` behaviour and handle specific system messages that are not exposed to the programmer. This is different from the way in which one process links to another in raw Erlang, and because of this, we cannot mix the two mechanisms.

The `application` behaviour is used to package Erlang modules into reusable components. An Erlang system will consist of a set of loosely-coupled applications. Some are developed by the programmer or the open source community, and others will be part of the OTP distribution. The Erlang runtime system and its tools will treat all applications equally, regardless of whether they are part of the Erlang distribution or not.

There are two kinds of applications. The most common form of applications, called normal applications, will start the supervision tree and all of the relevant static workers. Library applications such as the Standard Library, which come as part of the Erlang distribution, contain library modules but do not start the supervision tree. This is not to say that the code may not contain processes or supervision trees. It just means they are started as part of a supervision tree belonging to another application.

For more information on behaviours not covered in this paper, we recommend the OTP Design Principles User's Guide, available in the documentation section of the http://erlang.org website.

Conclusions

The generic servers described in these lecture notes give an example of how OTP behaviours work. Behaviours we have not covered but which we briefly introduced in this chapter include finite state machines, event handlers, supervisors and special processes. All of these behaviour library modules have manual pages that you can reference. In addition, the Erlang documentation has a section on OTP design principles that provides more detailed explanations and examples.

Workers and supervisors create supervision trees which when packaged in applications give software architects a generic and powerful approach to packaging and deployment of software. The benefits are reduced code sizes, generic error handling and reuse of components, ensuring that you don't "reinvent the wheel" in writing Erlang solutions.