Distributed Programming with Erlang
A crash course

Cons T Åhs
cahs@cisco.com/cons@tail-f.com
@lisztspace

Tail-f is now part of Cisco.

Cons T Åhs

- Technical Leader at Cisco since two weeks ago
  - Developing network configuration and management tools in Erlang
- Previously
  - Senior developer/architect, Keeper of The Code at Klarna (probably Sweden's largest collection of Erlang developers)
  - Consultant; online poker, low level networking, medical imaging, graphics, finance, musical notation, compilers, real time video decoding, teaching..
  - Lecturer at Uppsala University, research & teaching; foundations, algorithms, functions, relations, objects, compilers, pragmatics, theory, theorem proving, formal program correctness..

Erlang - The Language

- Conceived at Ericsson
- Buzzword compliancy
  - Functional - no side effects
  - Robust - built for fault tolerance and high availability
  - Runs in a virtual machine (VM) called beam
  - Extremely lightweight processes - from 309 words
  - Easy to distribute among cores, VMs and machines
  - No shared memory between processes
  - Processes communicate asynchronously through mail boxes
  - OTP - Open Telecom Platform

A Functional Language

- Dynamically typed functional language
- No side effects; variables are bound once and the value can not be changed
  - trying to reassign a variable will crash the program
- Every expression computes a value
- Pattern matching provides parallel binding and compact programs (mixed blessing - beware!)
- Looks very much like Prolog
  - A function is determined by both name and arity
  - functions are divided in clauses
  - function bodies are sequences of expressions
- Includes the power of higher order functions and closures

Basic Workings

- The file `example.erl` holds module `example`
- The exported functions constitutes the interface of the module
- Access exported functions with `module:fun(<args>)`
- Erlang is started with `erl` presenting you with a basic REPL (read-eval-print loop)
  - enter expressions and see value
- Use `c/1` to compile a file
- Use `l/1` to load a compiled file [lowercase ‘L’]

Factorial

```
-module(fact).

-export([factorial/1]).

factorial(N) when N > 0 ->
  N*factorial(N-1);
factorial(0) -> 1.
```

```
-module(fact).

-export([factorial/1]).

factorial(N) -> factorial(N, 1).

factorial(N, F) when N > 0 ->
  factorial(N-1, F*N);
factorial(0, F) -> F.
```

- Stored in file `fact.erl` - an erlang module corresponds to a single file
- Only exported functions (`factorial/1`) are available externally
- Clauses are tested in order
- Clauses are separated with a semicolon
- Last clause ends with a period
- Variables starts with a uppercase character
- The expression after `when` is called a *guard* - limited set of operators allowed, not any arbitrary function call
- Why is the version on the right better?

Append lists

```
-module(append).

-export([append/2]).

append([], L) -> L;
append([X|Xs], L) -> [X|append(Xs, L)].
```

- List syntax
  - `[]` for empty list
  - `[Head | Tail]` pattern for head and tail of list
  - `[1, 2, 3]` list of three element
- Pattern matching can be used in clauses
  - Runtime error if there is no clause matching the call
- What's the complexity of this function?
- The builtin ++ operator does the same thing, so L1 ++ L2 appends the lists.

Usage

```
2> fact:factorial(10).
3628800
3> fact:factorial(100).
93326215443944152681699238856266700490715968264381621468
59296389521759999322991560894146397615651828625369792082
7223758251185210916864000000000000000000000000

7> append:append([1,2,3],[a,b,c]).
[1,2,3,a,b,c]
8>
```

- Function call uses both module and function name
- Erlang has bignums, i.e., arbitrarily large integers
- Lists with mixed types are allowed
- Erlang is not a typed language
  - Type errors not caught at compile time

Tuples

```
-module(tuples).

-export([build/2, first/1, second/1]).

build(X, Y) -> {X, Y}.

first({X, _}) -> X.

second({_, Y}) -> Y.
```

- You can group N (N≥0) things in a tuple
- Pattern matching can be done on tuples as well as lists
- _ is an anonymous variable, i.e., a placeholder for an ignored value
  - This extends to any variable starting with an underscore, e.g, _Foo

Conditional computation

- Pattern matching together with clauses is one way of doing conditional computation.
- The traditional way in a functional language is to supply a built in construct
  - C -> E1; E2
  - C is an arbitrary expression that evaluates to true or false
  - If C evaluates to true, E1 is evaluated and the value returned
  - If C evaluates to false, E2 is evaluated and the value returned
- Why is this described with the term “construct” instead of “function”?
- Some languages got this extremely right from the start, Erlang did not..

Conditional - case

```
case lists:member(3, L) of
  true -> ... ;
  false -> ...
end

case foo(X, Y, Z) of
  ok -> ... ;
  [] -> ... ;
  {U, V} -> ...U..V
end
```

- Evaluate expression and match different results
- Cases are separated with semicolon
- Last case clause does not end with semicolon (or period)
- An `end` marks the end of the case clauses
- The result of the expression can be any type, which is reflected in the case clauses
- Variables can be bound in the patterns

Conditional - if

```
if
  integer(X) -> ... ;
  tuple(X) -> ... ;
  N > 0 -> ...
  true -> ...
end
```

- This is **not** a traditional function if!
- One can **not** write arbitrary expressions in the conditional, only *guards*
- Erlang's `if` is generally considered to be broken and you'll actually very seldom see it used in real programs.
- `case` and/or pattern matching is used instead
- A guard is an expression consisting only of operators and built in functions
  - A construct to make computation efficient

Compute length of list

```
-module(ex1).

-export([ rlen/1
        , tlen/1
        ]).

%% Ordinary recursive definition
rlen([])     -> 0;
rlen([_| L]) -> 1 + rlen(L).

%% Tail recursive definition
tlen(L) ->
  tlen(L, 0).

%% Tail recursive help function
tlen([], N)     -> N;
tlen([_| L], N) -> tlen(L, N+1).
```

Data representation

- Data is built from numbers, atoms, tuples and lists
  - `11, 42, 4711, 3.141692657`
  - `foo, cisco, tail_f, last_name, false`
  - `{foo, 12}`
  - `{ray, {vec, 0.0, 1.0, 1.2}, {vec, 1, 1, 1}}`
  - `[foo, bar, baz]`
  - `[{object, 12}, wall, {true, 42}]`
- Strings are just lists of characters (!)
- There is some support for abstraction in the form of records
- Also, opaque data such as pids, binaries, refs

Quirk: No Strings(!)

```
9> append:append("no ", "strings").
"no strings"
10> [97, 98, 99].
"abc"
11>
```

- The normal string notation is just syntactic sugar for a list of character codes
- Lists of integers that (seem to) represent characters are printed as strings
- All list operations can be used on strings

Records

```
-record(person, {name, age=0, length}).

mk_person(Name) -> #person{name=Name}.

mk_person(Name, Age) ->
  #person{name=Name, age=Age}.

get_name(#person{name=Name}) -> Name.

get_age(Person) -> Person#person.age.

change_age(Person, Age) ->
  Person#person{age=Age}.
```

- Syntactic sugar for tuples with first component being the name of the record
- A somewhat abstract representation - changes in representation can be hidden
- Record syntax can be used in pattern matching
- Records were added to the language as an afterthought

Insert into ordered tree

```
-module(ex2).

-export([cinsert/2]).

-record(tree, {info, left=empty, right=empty}).

cinsert(E, empty) -> #tree{info = E};
cinsert(E, T = #tree{info = E}) -> T;
cinsert(E, T = #tree{info = I}) when E < I ->
  T#tree{left = cinsert(E, T#tree.left)};
cinsert(E, T = #tree{info = I}) when E > I ->
  T#tree{right = cinsert(E, T#tree.right)}.
```

Abstract insert

```
-module(ex3).

-export([ empty_tree/0, insert/2]).

-record(tree, {info, left=empty, right=empty}).

empty_tree()               -> empty.
tree_info(#tree{info = I}) -> I.
tree_left(#tree{left = L}) -> L.
tree_right(#tree{left = R}) -> R.

is_empty_tree(empty)    -> true;
is_empty_tree(#tree{}) -> false.

mk_node(E)               -> #tree{info = E}.
mk_tree(E, Left, Right) -> #tree{info = E, left = Left, right = Right}.

insert(E, Tree) ->
  case is_empty_tree(Tree) of
    true  -> mk_node(E);
    false ->
      I = tree_info(Tree),
      if E == I -> Tree;
         E < I ->
          mk_tree(I, insert(E, tree_left(Tree)), tree_right(Tree));
         true ->
          mk_tree(I, tree_left(Tree), insert(E, tree_right(Tree)))
      end
  end.
```

Similar syntax, different meaning

Are these all the same?

No.
The types are different

```
is_empty_tree(empty)   -> true;
is_empty_tree(#tree{}) -> false.
```

```
empty | #tree -> true | false
```

```
is_empty_tree(empty) -> true;
is_empty_tree(_)     -> false.
```

```
any() -> true | false
```

```
is_empty_tree(Tree) -> Tree == empty.
```

```
any() -> true | false
```

```
is_empty_tree(Tree) -> Tree = empty.
```

```
empty -> empty
```

The last two shows the difference between binding and matching

Binding and matching

```
foo1(N) ->
  X = N,
  X = 1.

foo2(N) ->
  X = N,
  X == 1.
```

- Variables are single assignment, so the first occurrence of a variable will bind it
- In subsequent occurrences, the bound value is used and can not be changed
- The = operator does bind **and** matching (of patterns) and can fail, i.e., generate a runtime error (if the variable already has a value)
- The == operator does **only** matching (no binding) and returns true or false.
- What is the difference between `foo1/1` and `foo2/1`?

Local variables, scope

```
f(X, Y) ->
    A = X+Y,
    B = X-Y,
    {A, B}.

g(T) ->
    M = case T of
        {N} -> true;
        {N, _} -> false
       end,
    {M, N}.
```

- The scope of a local variable binding is the rest of the clause
- This is true even if a variable is introduced by pattern matching in a case clause

Higher order functions

- Functions are first class citizens
  - a variable can be bound to a function
  - a function can be the result of a computation
  - a function can be passed as an argument

```
-module(ex4).

-export([ sorttuples/1
        ]).

sorttuples(Tuples) ->
  Num = fun({_, N}) -> N end,
  Cmp = fun(T1, T2) -> Num(T1) < Num(T2) end,
  lists:sort(Cmp, Tuples).
```

Higher order functions

```
hof() ->
  F = fun(X) -> X * X + 1 end,
  L = lists:map(F, [1, 2, 3],

  G = fun([])     -> nil;
         ([_|_]) -> cons
      end,

  Y = G(L),
  Y == nil.
```

- Syntax for anonymous functions is rather verbose
- Anonymous functions can have several clauses and use pattern matching
- A variable can be bound to a function
- Apply the function by using the variable instead of a function name
  - Erlang got this right!
- What is the value of `hof()`?

Scoping revisited

- The scope of a variable binding is the rest of the function clause
  - An expression can only access variables bound before the expression
  - It is not possible to write a local recursive function in the "ordinary" way

```
no(N) ->
  G = fun(0) -> 1;
         (N) -> N*G(N-1)
      end,

  G(N).
```

- It is possible to write a "local recursive" function using higher order functions
  - Observe that G inside is "just" a function variable so it has to be passed to the function
  - This is a good exercise!
  - Write factorial in this way.

Higher order functions

```
make_adder(N) ->
  fun(X) -> X + N end.

inclist(L) ->
  lists:map(make_adder(3), L).

whatlist(L) ->
  lists:map(fun make_adder/1, L).

what(L, V) ->
  lists:map(fun(F) -> F(V) end, L).
```

- A function can be returned
- Notation for passing a named function as an argument
- Describe the functions `inclist/1`, `whatlist/1` and `what/2`

Higher order functions

```
cumbersome(M) ->
  MakeAdder = fun(N) ->
                fun(X) -> X + N end
              end,
  (MakeAdder(3))(M).
```

- Making curried functions suitable for partial application is possible, but quickly becomes a bit difficult to read.
- This is much easier in languages designed for this from the start.

Digression on closures

```
make_adder(N) ->
  fun(X) -> X + N end.

make_what(M) ->
  fun() -> fibonacci(M) end.

do_it(D) ->
  D().
```

- We have the cool feature of being able to return a closure, i.e., a function and the environment it was defined in.
- What does `make_what/1` do?
  - Returns a function of no (?) argument.
  - It delays a computation!
  - The body is evaluated only when we apply the result (of make_what/1) to ().
- We can thus save and represent a computation and do it later.

Variables can hold anything

```
-module(sequences).
-export([plus/2, minus/2]).

plus(X, Y) -> X ++ Y.
minus(X, Y) -> X -- Y.
```

```
-module(numbers).
-export([plus/2, minus/2]).

plus(X, Y) -> X + Y.
minus(X, Y) -> X - Y.
```

```
-module(eval).
-export([eval/4]).

eval(M, F, A1, A2) ->
    M:F(A1, A2).
```

```
10> eval:eval(sequences, plus, [1,2,3], [a,b,c]).
[1,2,3,a,b,c]
11> eval:eval(numbers, plus, 4, 7).
11
12>
```

Variables can hold anything

- A variable can be bound to
  - ordinary values and functions (no surprise)
  - function *names*
  - *modules*
- This means you can send a whole module M as an argument to another function and the receiving function then calls known functions in M.
  - Is this useful?
  - Yes!
- It also means that given a module you can vary the actual function that is called by passing the *name* in a variable.
  - Is this useful?
  - Possibly.
- Both variations lead to the possibility to map, e.g., user input directly to Erlang modules and functions at runtime.
  - Great way to make a really insecure system!

Variables can hold anything

- We had two modules which exported the same function names and arities
  - They thus have the same interface!
  - This concept exists in Erlang, but has the name *behaviour*
  - It can be used in the same way as in, e.g., Java by providing several different implementations of the same (abstract) interface
  - A very commonly used behaviour is the gen_server (for generic server)
  - You provide the details and a generic server takes care of the generic parts.

BIFs (Built In Functions)

- BIFs exist to provide functionality that can't be done in pure Erlang
  - interface with the real world for things like date, time and low level file system access
  - conversion between primitive types such as
    - atom_to_list (convert an atom to a "string")
    - list_to_atom (convert a "string" to a (new) atom)
    - etc
- There might also be BIFs for functions that can be implemented in Erlang, but a BIF will do it faster.
- Read documentation!

Standard Libraries

- Erlang comes with a large set of standard libraries, e.g,
  - list function
  - dictionaries of varying representation
  - ets, dets - term storage, either in memory or on disk
  - mnesia - database built on top of dets
  - etc
- Read the documentation

Concurrent and distributed programming

- With concurrent programming troubles form when you have a shared <u>and</u> mutable state.
- Problem typically solved by using synchronisation with locks
  - Complicated - you have to know when to lock
  - Can lead to more problems - performance degradation
  - Cooperative model - all parts of the program must agree
- Take away one and your on safe ground.
- Erlang takes away both!

No shared state, no mutable state

- Each process has a state of its own, or rather a sequence of states; possibly a new state after receiving a message
- Each process has a private heap
- Each process has a message queue (the implementation handles these)
- Processes can not share state, even when they live in the same VM.
  - All communication must be done with messages.
  - messages are copied between processes

No shared state

- Why?
  - Background (telecom switches) with a large number of small and short lived processes
  - When a process dies there is no risk reclaiming the whole process
  - No other process can access the memory it used
  - Nothing happens if you send a message to a dead pid
  - The dead process can not reference the memory of another process
  - Leads to robustness

Keeping state in a process

- Real world computations need state
- State is encoded in a process that reacts to messages
  - init state
  - wait for message
  - compute new state from message and existing state
  - loop

```
start() -> actor(init_state()).

actor(State) ->
  actor(process_message(get_msg(), State)).
```

- start the actor and send messages to it

Managing Processes

- Three basic primitives are used to handle processes
- Create process - returns pid (process id)

```
spawn(Function) or spawn(M, F, Args)
```

- Send a message - returns Msg

```
Pid ! Msg
```

- Receive a message from the message queue (the process will wait if there is no message) - returns value of chosen expression

```
receive
  Pattern1 -> Expr1;
  Pattern2 -> Expr2;
  ...
end
```

Simple Message Passing

- Note that you have to set up the actual protocol yourself
- If you want a reply, a sent message should include a return address
- This goes for the reply as well - the original sender might want to know who sent the reply
- This might also apply to request identifiers so a more general request would contain both a return address and an identifier
- Given a simple and light weight protocol you can build a more complicated protocol (with delivery guarantees) upon it, but not the other way round.

Selective receive

- Note that a receive will wait until it finds a message matching the pattern
  - Messages might not be processed in the order they come
  - This can be expensive since the message queue has to be searched

```
receive
  foo -> f(..)
end,
receive
  bar -> g(..)
end
```

Receiving messages

```
foobar() ->
  F = fun(Msg) ->
   {message_queue_len, L} = process_info(self(), message_queue_len),
   io:format("Msg: ~p (~p)~n", [Msg, L])
  end,

  receive M0=foo -> F(M0) end,
  receive M1=bar -> F(M1) end,
  foobar().
```

- A `receive` will wait until a message matching a specified pattern is in the queue.
- Messages are processed in an order specified by the `receives` in the process
- Messages are thus not necessarily processed in the order they arrive
- The code
  - reports queue length when acting on a message
  - messages are processed in the sequence `foo, bar, foo, bar, ..`
  - Note use of binding pattern in `receive`
  - Why can't we have the same variable in both `receives`

Example

```
start() -> server(0).

server(Count) ->
  NewCount = receive
               {report, Pid} ->
                 Pid ! Count,
                 Count;
               _Msg -> Count + 1
        end,
  server(NewCount).

32> P = spawn(fun simple:start/0).
<0.110.0>
33> P!foo.
foo
34> P!foo.
foo
35> P!foo.
foo
36> P!{report, self()}.
{report,<0.88.0>}
37> receive M -> M end.
3
```

Efficient computation through memoisation

- Consider a computationally intensive function
  - Fibonacci, Ackermann, ..
- Instead of computing the value each time, one can remember the values and serve them when a new request comes
  - If we know the value, return it
  - Otherwise, compute it, remember it, return it
- It's actually a cache!
- The cache (a mapping from argument(s) to value) is encoded in the state of a process

Efficient computation through memoisation

```
-module(ex5).

-export([ fib/1, fibfun/0]).

fib(0) -> 1;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).

fibfun() ->
  Cache = dict:new(),
  Pid = spawn(fun() -> loop(Cache) end),
  fun(N) ->
      Pid ! {self(), N},
      receive
        V -> V
      end
  end.

loop(Cache) ->
  receive
    {Pid, N} ->
      case dict:find(N, Cache) of
        {ok, Value} ->
          NewCache = Cache;
        error ->
          Value = fib(N),
          NewCache = dict:store(N, Value, Cache)
      end,
      Pid ! Value,
      loop(NewCache)
  end.
```

Distribution made easy

- Distribute work load among a number of workers
- Input
  - the work to be done, a queue of tasks
  - the workers that performs the work (pids)
- What is specific for each problem?
  - How to get a chunk of work from the queue
  - How to combine results from a single worker with the result from the others

Distribution made easy

- We're done when the queue is empty <u>and</u> we have no active workers.
- We wait for a worker to return a result when the queue is empty <u>or</u> we have no passive workers
- We activate a worker when the queue is non empty and we have passive workers.
- Initial state is a queue of work, no active workers and a collection of passive workers.

Distribution made easy

```
sequential(L) -> lists:filter(fun is_prime/1, L).

process_work([], [], _, State) -> State;
process_work(Work, Active, Passive, State)
  when Work =:= []; Passive =:= [] ->
  receive {Worker, M} ->
      process_work(Work, lists:delete(Worker, Active),
                   [Worker | Passive], add_result(State, M))
  end;
process_work(Work, Active, [Worker | Passive], State) ->
  {Chunk, Rest} = get_chunk(State, Work),
  Worker ! {self(), Chunk},
  process_work(Rest, [Worker | Active], Passive, State).

worker() ->
  receive {Pid, Work} ->
      Pid ! {self(), sequential(Work)},
      worker()
  end.
```

Linking processes

- Send a message (with `Pid ! Message`) returns the message.
  - This happens even if the process has died
  - No delivery receipt
  - if `process_info(Pid) == undefined` the process is not alive
  - querying the process status is impractical
- A process will run until it
  - terminates normally
  - is killed by someone else
  - is killed by an accident
- A system with several processes will not work if one process ceases to exist
  - default is that process death is ignored - no one cares
  - The rest of system needs to know about the death of other processes
  - Possible actions
    - take down other processes
    - restart dead process
    - restart several other processes

Linking processes

- Processes can be tied together with *links*
- Two (of several) ways to create links
  - `link(Pid)` - link current process with `Pid`
  - `spawn_link(Fun)`- create new process and link it with current process
- Linking processes means linking their destiny
  - Links are bidirectional
  - Without additional considerations in place, a process P_0 linked to P_1 will terminate if P_1 terminates (and vice versa)
  - This is (slightly) better since we'll have no silent sending of messages to dead processes.
- A process that dies/exits will send a signal to linked processes and they will react by dying as well.

Linking processes

```
failing() ->
  receive
    X ->
      io:format("failing, msg: ~p~n", [X]),
      X=elrang,
      failing()
    end.
```

```
124> f(P), P = spawn(fun() -> linking:failing() end).
<0.300.0>
125> P!foo.
failing, msg: foo
foo

=ERROR REPORT==== 4-Nov-2012::09:57:41 ===
Error in process <0.300.0> with exit value:
{{badmatch,elrang},[{linking,failing,0}]}
```

Linking processes

```
parent() ->
  Child = spawn_link(fun() -> failing() end),
  receive
    M ->
      io:format("Parent, msg: ~p~n", [M]),
      Child ! M,
      parent()
  end.
```

```
f(P), P = spawn(fun() -> linking:parent() end).
<0.314.0>
132> P!bar.
Parent, msg: bar
bar
failing, msg: bar
133>
=ERROR REPORT==== 4-Nov-2012::10:03:17 ===
Error in process <0.315.0> with exit value:
{{badmatch,elrang},[{linking,failing,0}]}

P!hello.
hello
134>
```

Linking processes

- Much better is to be made aware of a linked process being in trouble
- Catch the signal, convert it to a message and act upon it.
- This is the base for building robust systems that act upon failures

```
responsible_parent() ->
  process_flag(trap_exit, true),
  care_for().

care_for() ->
  Child = spawn_link(fun() -> failing() end),
  care_for(Child).

care_for(Child) ->
  receive
    {'EXIT', Child, Why} ->
       io:format("child died (reason: ~pn), restart it~n", [Why]),
       care_for();
    M ->
       io:format("Parent, msg: ~p~n", [M]),
       Child ! M,
       care_for(Child)
  end.
```

Behaviours

- A *behaviour* in Erlang specifies the *interface* of a module
  - A module *must* implement the functions specified by the behaviour
  - It can implement and export more functions
  - A module that implements a behaviour can then be passed to a generic module expecting that behaviour
  - This can also rather easily be implemented using higher order functions

Behaviours

- The actual behaviour is specified by the function `behaviour_info/1`
- It should return a list of tuples `{functionname, arity}`
- The actual implementation making use of the implementation can be in the same module defining the behaviour or in another module.
- There is no checking that the module supplied actually implements the behaviour - this is discovered at runtime.
- Example: implement a generic module for caching the values of a (pure) function call. Since the actual computation might take a long time, we want to avoid computing the function several times.
- General idea:
  - Receive a "function call"
  - Check the cache if we already have computed the value
    - If so, return the value (no change in the cache)
    - If not, compute the value, add it to the cache and return the value

```
-module(cachefun).

-export([init/1 , behaviour_info/1]).

behaviour_info(callbacks) -> [{compute, 1}];
behaviour_info(_) -> undefined.

init(Module) ->
  Cache = dict:new(),
  Pid = spawn(fun() -> loop(Cache, Module) end),
  fun(X) ->
      Pid ! {self(), X},
      receive V -> V end
  end.

loop(Cache, Module) ->
  receive {Pid, Arg} ->
   case dict:find(Arg, Cache) of
     {ok, Value} ->
       NewCache = Cache;
     error ->
       Value = Module:compute(Arg),
       NewCache = dict:store(Arg, Value, Cache)
   end,
      Pid ! Value,
      loop(NewCache, Module)
  end.
```

Behaviours

- `fibfun()` returns a function
- `?MODULE` is a macro returning the module name

```
-module(fibcache).

-behaviour(cachefun).

-export([compute/1, fibfun/0]).

fibfun() -> cachefun:init(?MODULE).

compute(N) -> fib(N).

fib(0) -> 0;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
```

```
3> F= fibcache:fibfun().
#Fun<cachefun.1.45378360>
4> F(40).
```

Standard behaviours

- `gen_server` - implements a generic server, supporting
  - request/response (synchronous calls)
  - commands (requests without response, or asynchronous calls)
  - code upgrade
  - You implement the specific details for handling state and responding to the calls, the generic server takes care of the rest
- `supervisor` - implements generic functions for supervising processes, i.e., how the different processes should react when process die etc.
- `gen_fsm` - finite state machine; you code the states, events and transistions and the generic machine takes care of the rest.

Code loading

- One core feature of Erlang is the ability to load new code during runtime
- To cater for scenarios where you "long" running processes Erlang actually supports holding two versions (current and old) of a module at a given time.
- When a new version is loaded the old is thrown away, the (previously) current becomes the old and newly loaded becomes the current.
- This works for external calls, i.e., a module calls another using a module prefix.
- For an internal call a name always refers to the code version in the module
  - a process holding a reference to an old module might fail due to the code being unloaded and thrown away
- This is "solved" by always calling with the module prefix, but it also means that the function has to be exported.
  - the current (newest) version is always called

```
-module(server).

-export([loop/1]).

loop(State) ->
  <wait for messages and compute new state>,
  server:loop(NewState).
```

Binaries

- The telecom world is full of protocols, often at a very low level, i.e., 3 bits for this, followed by 7 bits for that etc.
- Erlang makes it very easy to manipulate bit strings, treating them in a very nice abstract manner.
- External syntax `<<..>>` where .. is a sequence of bit field specifiers
- A binary is a datatype in the same way as numbers, terms, lists etc
  - integers must be converted to and from binaries
- Instead of masking and shifting one can extract bitfields through matching
- Similarly, one can construct a binary the same way.

```
decode_parts(<<T:1, F:3, U:2, S:2>>) ->
  {T==1, F, U, S}.

encode_parts({Flag, F, U, S}) ->
  T = if Flag -> 1;
         true -> 0
      end,
  <<T:1, F:3, U:2, S:2>>.
```

Binaries

- Decoding an IP (V4) datagram

```
ip_datagram(Dgram) ->
  Size = byte_size(Dgram),
  case Dgram of
    <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
      ID:16, Flgs:3, FragOff:13,
      TTL:8, Proto:8, HdrChkSum:16,
      SrcIP:32,
      DestIP:32, RestDgram/binary>> when HLen>=5, 4*HLen=<Size ->
      OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
      <<Opts:OptsLen/binary,Data/binary>> = RestDgram,
      ...
  end.
```

Storage and Persistence

- Any real life application will have the need to handle larger amounts of data
  - in memory (with pragmatic access)
  - persistently (still there after a restart)
  - efficient access (constant)
  - distributed
- Erlang provide several options
  - process dictionary - “global storage” for a process (limited use)
  - ets - erlang term storage, table based, in memory, belongs to a process
  - dets - disk based ets, persistent (similar to ets in operations, but slower)
  - mnesia - database built on which support transactions and distribution

Erlang Summary

- Untyped language with a functional core.
- Evolved rather than designed.
- Designed for fault tolerance, distribution and robustness.
- Excellent handling of processes.
- Not an excellent language for abstraction and “normal” software engineering.
- Not so well designed in terms of syntax and some semantics.
- Some rather horrible constructions.

- Uncovered topics
  - most of the standard libraries (otp)
  - tools surrounding development and releases
  - behaviours, generic servers
  - lots of details

More about Erlang

- Covered the basics of Erlang and distributed and concurrent programming
- OTP, Supervisors, behaviours, gen_server, rebar, eunit, proper, dialyzer, standard libraries, persistence in various forms, bit syntax, code loading, actual side effects ..
- Good book
  - Erlang and OTP in Action by Martin Logan, Eric Meritt, Richard Carlsson.