



Erlang: An Overview in Four Parts

Part 1 – Sequential Erlang

Thanks to Richard Carlsson for the original version of many of the slides in this part



Erlang Buzzwords

- Functional (strict)
- Single-assignment
- Dynamically typed
- Concurrent
- Distributed
- Message passing
- Soft real-time
- Fault tolerant
- Shared-nothing
- Automatic memory management (GC)
- Virtual Machine (BEAM)
- Native code (HiPE)
- Dynamic code loading
- Hot-swapping code
- Multiprocessor support
- OTP (Open Telecom Platform) libraries
- Open source

Background

- Developed by Ericsson, Sweden
 - Experiments 1982-1986 with existing languages
 - Higher productivity, fewer errors
 - Suitable for writing (large) telecom applications
 - Must handle concurrency and error recovery
 - No good match - decided to make their own
 - 1986-1987: First experiments with own language
 - Erlang (after Danish mathematician A. K. Erlang)
 - 1988-1989: Internal use
 - 1990-1998: Erlang sold as a product by Ericsson
 - Open Source (MPL-based license) since 1998
 - Development still done by Ericsson



Erlang at Uppsala University

- High Performance Erlang (HiPE) research group
 - Native code compiler
 - back-ends: SPARC, x86, x86_64, PowerPC, PowerPC-64, ARM
 - Program analysis and optimization
 - Runtime system improvements
 - Language development and extensions
 - Programming and static analysis tools
- Most results from the HiPE project have been included in the official Erlang distribution



Hello, World!

```
%% File: hello.erl
-module(hello) .
-export([run/0]) .

-spec run() -> 'ok' .
run() -> io:format("Hello, World!\n") .
```

- '%' starts a comment
- '.' ends each declaration
- Every function must be in a module
 - One module per source file
 - Source file name is module name + ".erl"
- ':' used for calling functions in other modules



Running Erlang

```
$ erl
Erlang (BEAM) emulator version 5.10.3

Eshell V5.10.3 (abort with ^G)
1> 6*7.
42
2> halt().
$
```

- The Erlang VM emulator is called 'erl'
- The interactive shell lets you write any Erlang expressions and run them (must end with '.'')
- The "1>", "2>", etc. is the shell input prompt
- The "halt()" function call exits the emulator



Compiling a module

```
$ erl
Erlang (BEAM) emulator version 5.10.3

Eshell V5.10.3 (abort with ^G)
1> c(hello) .
{ok,hello}
2>
```

- The “`c(Module)`” built-in shell function compiles a module and loads it into the system
 - If you change something and do “`c(Module)`” again, the new version of the module will replace the old
- There is also a standalone compiler called “`erlc`”
 - Running “`erlc hello.erl`” creates “`hello.beam`”
 - Can be used in a normal Makefile



Running a program

```
Eshell V5.10.3 (abort with ^G)
1> c(hello) .
{ok,hello}
2> hello:run() .
Hello, World!
ok
3>
```

- Compile all your modules
- Call the exported function that you want to run, using “`module:function(...)`.”
- The final value is always printed in the shell
 - “ok” is the return value from `io:format(...)`



A recursive function

```
-module(factorial) .  
-export([fact/1]) .  
  
-spec fact(non_neg_integer()) -> pos_integer() .  
fact(N) when N > 0 ->  
    N * fact(N-1) ;  
fact(0) ->  
    1 .
```

- Variables start with upper-case characters!
- ';' separates function clauses; last clause ends with '.'
- Variables are local to the function clause
- Pattern matching and 'when' guards to select clauses
- Run-time error if no clause matches (e.g., $N < 0$)
- Run-time error if N is not an integer



Tail recursion with accumulator

```
-module(factorial) .  
-export([fact/1]) .  
  
-spec fact(non_neg_integer()) -> pos_integer() .  
fact(N) -> fact(N, 1) .  
  
fact(N, Fact) when N > 0 ->  
    fact(N-1, Fact*N) ;  
fact(0, Fact) ->  
    Fact .
```

- The *arity* is part of the function name: `fact/1`≠`fact/2`
- Non-exported functions are local to the module
- Function definitions cannot be nested (as in C)
- Last call optimization is performed: the stack does not grow if the result is the value of another function call



Recursion over lists

```
-module(list) .  
-export([last/1]) .  
  
-spec last([T]) -> T.  
last([Element]) -> Element;  
last([_|Rest]) -> last(Rest) .
```

- Pattern matching selects components of the data
- “_” is a “don't care”-pattern (not a variable)
- “[Head|Tail]” is the syntax for a single list cell
- “[]” is the empty list (often called “nil”)
- “[X, Y, Z]” is a list with exactly three elements
- “[X, Y, Z|Tail]” has three or more elements



List recursion with accumulator

```
-module(list) .  
-export([reverse/1]) .  
  
-spec reverse([T]) -> [T] .  
reverse(List) -> reverse(List, []).  
  
reverse([Head|Tail], Acc) ->  
    reverse(Tail, [Head|Acc]);  
reverse([], Acc) ->  
    Acc.
```

- The same syntax is used to *construct lists*
- Strings are simply lists of Unicode characters
 - "Hello" = [\$H, \$e, \$l, \$l, \$o] = [72,101,108,108,111]
 - "" = []
- All list functions can be used on strings

12345

-9876

16#ffff

2#010101

\$A

0.0

3.1415926

6.023e+23

- Arbitrary-size integers (but usually just one word)
- #-notation for base-N integers
- \$-notation for character codes (ISO-8859-1)
- Normal floating-point numbers (standard syntax)
 - cannot start with just a '.', as in e.g. C



Atoms

```
true           % Boolean
false          % Boolean
ok             % used as "void" value
hello_world
doNotUseCamelCaseInAtoms
'This is also an atom'
'foo@bar.baz'
```

- Must start with lower-case character or be quoted
- Single-quotes are used to create arbitrary atoms
- Similar to hashed strings
 - Use only one word of data (just like a small integer)
 - Constant-time equality test (e.g., in pattern matching)
 - At run-time: `atom_to_list(Atom)`, `list_to_atom(List)`

```
{ }  
{42}  
{1,2,3,4}  
{movie, "Yojimbo", 1961, "Kurosawa"}  
{foo, {bar, X},  
      {baz, Y},  
      [1,2,3,4,5]}
```

- Tuples are the main data constructor in Erlang
- A tuple whose 1st element is an atom is called a *tagged tuple* - this is used like constructors in ML
 - Just a convention – but almost all code uses this
- The elements of a tuple can be any values
- At run-time: `tuple_to_list(Tup)`, `list_to_tuple(List)`



Other data types

- Functions
 - Anonymous and other
- Bit streams
 - Sequences of bits
 - `<<0,1,2,...,255>>`
- Process identifiers
 - Usually called 'Pids'
- References
 - Unique “cookies”
 - `R = make_ref()`
- No separate Booleans
 - atoms `true/false`
- Erlang values in general are often called “terms”
- All terms are ordered and can be compared with `<`, `>`, `==`, `:=:`, etc.



Type tests and conversions

```
is_integer(X)
is_float(X)
is_number(X)
is_atom(X)
is_tuple(X)
is_pid(X)
is_reference(X)
is_function(X)
is_list(X)    % [] or [_|_]
```

```
atom_to_list(A)
list_to_tuple(L)
binary_to_list(B)
```

```
term_to_binary(X)
binary_to_term(B)
```

- Note that `is_list` only looks at the first cell of the list, not the rest
- A list cell whose tail is not another list cell or an empty list is called an “improper list”.
 - Avoid creating them!
- Some conversion functions are just for debugging: avoid!
 - `pid_to_list(Pid)`



Built-in functions (BIFs)

```
length(List)
tuple_size(Tuple)
element(N, Tuple)
setelement(N, Tuple, Val)
```

```
abs(N)
round(N)
trunc(N)
```

```
throw(Term)
halt()
```

```
time()
date()
now()
```

```
self()
spawn(Function)
exit(Term)
```

- Implemented in C
- All the type tests and conversions are BIFs
- Most BIFs (not all) are in the module “erlang”
- Many common BIFs are auto-imported (recognized without writing “erlang: . . .”)
- Operators (+, -, *, /, ...) are also really BIFs



Standard libraries

Application Libraries

- erts
 - erlang
- kernel
 - code
 - file, filelib
 - inet
 - os
- stdlib
 - lists
 - dict, ordict
 - sets, ordsets, gb_sets
 - gb_trees
 - ets, dets

- Written in Erlang
- “Applications” are groups of modules
 - Libraries
 - Application programs
 - Servers/daemons
 - Tools
 - GUI system (gs, wx)

Expressions

```
%% the usual operators
```

```
(X + Y) / -Z * 10 - 1
```

```
%% boolean
```

```
X and not Y or (Z xor W)
```

```
(X andalso Y) orelse Z
```

```
%% bitwise operators
```

```
((X bor Y) band 15) bsl 2
```

```
%% comparisons
```

```
X /= Y           % not !=
```

```
X =< Y           % not <=
```

```
%% list operators
```

```
List1 ++ List2
```

- Boolean and/or/xor are *strict* (always evaluate both arguments)
- Use `andalso/orelse` for short-circuit evaluation
- “`=: =`” for equality, not “`=`”
- We can always use parentheses when not absolutely certain about the precedence

Fun expressions

```
F1 = fun () -> 42 end
42 = F1 ()

F2 = fun (X) -> X + 1 end
42 = F2 (41)

F3 = fun (X, Y) ->
      {X, Y, F1}
      end

F4 = fun ({foo, X}, Y) ->
        X + Y;
      ({bar, X}, Y) ->
        X - Y;
      (_, Y) ->
        Y
      end

F5 = fun f/3

F6 = fun mod:f/3
```

- Anonymous functions (lambda expressions)
 - Usually called “funs”
- Can have several arguments and clauses
- All variables in the patterns are *new*
 - *All variable bindings in the fun are local*
 - *Variables bound in the environment can be used in the fun-body*



Pattern matching with '='

```
Tuple = {foo, 42, "hello"},  
{X, Y, Z} = Tuple,
```

```
List = [5, 5, 5, 4, 3, 2, 1],  
[A, A | Rest] = List,
```

```
Struct = {foo, [5,6,7,8], {17, 42}},  
{foo, [A|Tail], {N, Y}} = Struct
```

- Successful matching binds the variables
 - But only if they are not already bound to a value!
 - A new variable can also be repeated in a pattern
 - Previously bound variables can be used in patterns
- Match failure causes runtime error (badmatch)

Case switches

```
case List of
  [X|Xs] when X >= 0 ->
    X + f(Xs);
  [_X|Xs] ->
    f(Xs);
  [] ->
    0;
  _ ->
    throw(error)
end
```

```
%% boolean switch:
case Bool of
  true -> ... ;
  false -> ...
end
```

- Any number of clauses
- Patterns and guards, just as in functions
- ';' separates clauses
- Use “_” as catch-all
- Variables may also begin with underscore
 - Signals “I don't intend to use the value of this variable”
 - Compiler won't warn if this variable is not used
- OBS: Variables may be already bound in patterns!

If switches and guard details

```
if
  0 =< X, X < 256 ->
    X + f(Xs);
  true ->
    f(Xs)
end
```

The above construct is better written as

```
case 0 =< X and X < 256 of
  true ->
    X + f(Xs);
  false ->
    f(Xs)
end
```

- Like a case switch without the patterns and the “when” keyword
- Need to use “true” as catch-all guard (Ugly!)
- Guards are special
 - Comma-separated list
 - Only specific built-in functions (and all operators)
 - No side effects



List comprehensions

```
%% map
[f(x) || x <- List]

%% filter
[X || X <- Xs, X > 0]
```

```
Eshell v5.10.3 (abort ...^G)
1> L = [1,2,3].
[1,2,3]
2> [X+1 || X <- L].
[2,3,4]
3> [2*X || X <- L, X < 3].
[2,4]
4> [X+Y || X <- L, Y <- L].
[2,3,4,3,4,5,4,5,6]
```

- Left of the “||” is an *expression template*
- “Pattern <- List” is a *generator*
 - Elements are picked from the list in order
- The other expressions are *Boolean filters*
- If there are multiple generators, you get all combinations of values



List comprehensions: examples

```
%% quicksort of a list
qsort([]) -> [];
qsort([P|Xs]) ->
    qsort([X || X <- Xs, X =< P])
    ++ [P]    % pivot element
    ++ qsort([X || X <- Xs, P < X]).
```

```
%% generate all permutations of a list
perms([]) -> [[]];
perms(L) ->
    [[X|T] || X <- L, T <- perms(L -- [X])].
```

- Using comprehensions we get very compact code
...which sometimes can take some effort to understand
Try writing the same code without comprehensions



Bit strings and comprehensions

- Bit string pattern matching:

```
case <<8:4, 42:6>> of
  <<A:7/integer, B/bits>> -> {A,B}
end
```

```
case <<8:4, 42:6>> of
  <<A:3/integer, B:A/bits, C/bits>> -> {A,B,C}
end
```

- Bit string comprehensions:

```
<< <<x:2>> || <<x:3>> <= Bits, X < 4 >>
```

- Of course, one can also write:

```
[ <<x:2>> || <<x:3>> <= Bits, X < 4 ]
```



Catching exceptions

```
try
  lookup(X)
catch
  not_found ->
    use_default(X);
  exit:Term ->
    handle_exit(Term)
end

%% with 'of' and 'after'
try lookup(X, File) of
  Y when Y > 0 -> f(Y);
  Y -> g(Y)
catch
  ...
after
  close_file(File)
end
```

- Three classes of exceptions
 - throw: user-defined
 - error: runtime errors
 - exit: end process
 - Only catch throw exceptions, normally (implicit if left out)
- Re-thrown if no catch-clause matches
- “after” part is always run (side effects only)



Old-style exception handling

```
Val = (catch lookup(X)),  
  
case Val of  
  not_found ->  
    %% probably thrown  
    use_default(X);  
  {'EXIT', Term} ->  
    handle_exit(Term);  
  _ ->  
    Val  
end
```

- “catch Expr”
 - Value of “Expr” if no exception
 - Value X of “throw(X)” for a throw-exception
 - “{'EXIT', Term}” for other exceptions
- Hard to tell what happened (not safe)
- Mixes up errors/exits
- In lots of old code

Record syntax

```
-record(foo,  
      {a = 0 :: integer(),  
       b      :: integer()}).  
  
{foo, 0, 1} = #foo{b = 1}  
  
R = #foo{}  
{foo, 0, undefined} = R  
  
{foo, 0, 2} = R#foo{b=2}  
  
{foo, 2, 1} = R#foo{b=1, a=2}  
  
0 = R#foo.a  
undefined = R#foo.b  
  
f(#foo{b = undefined}) -> 1;  
f(#foo{a = A, b = B})  
  when B > 0 -> A + B;  
f(#foo{}) -> 0.
```

- Records are just a syntax for working with tagged tuples
- You don't have to remember element order and tuple size
- Good for internal work within a module
- Not so good in public interfaces (users must have same definition!)

Preprocessor

```
-include("defs.hrl").  
  
-ifndef(PI).  
-define(PI, 3.1415926).  
-endif.  
  
area(R) -> ?PI * (R*R).  
  
-define(foo(X), {foo,X+1}).  
  
{foo,42} = ?foo(41)  
  
%% pre-defined macros  
?MODULE  
?LINE
```

- C-style token-level preprocessor
 - Runs after tokenizing, but before parsing
- Record definitions often put in header files, to be included
- Use macros mainly for constants
- Use functions instead of macros if you can (compiler can inline)

Dialyzer: A defect detection tool

- A static analyzer that identifies discrepancies in Erlang code bases
 - code points where something is wrong
 - often a bug
 - or in any case something that needs fixing
- Fully automatic
- Extremely easy to use
- Fast and scalable
- Sound for defect detection
 - “Dialyzer is never wrong”



Dialyzer

- Part of the Erlang/OTP distribution since 2007
- Detects
 - Definite type errors
 - API violations
 - Unreachable and dead code
 - Opacity violations
 - Concurrency errors
 - Data races (`-Wrace_conditions`)
- Experimental extensions with
 - Stronger type inference: type dependencies
 - Detection of message passing errors & deadlocks





How to use Dialyzer

- First build a PLT (needs to be done once)

```
> dialyzer --build_plt --apps erts kernel stdlib
```

- Once this finishes, analyze your application

```
> cd my_app  
> erlc +debug_info -o ebin src/*.erl  
> dialyzer ebin
```

- If there are unknown functions, you may need to add more Erlang/OTP applications to the PLT

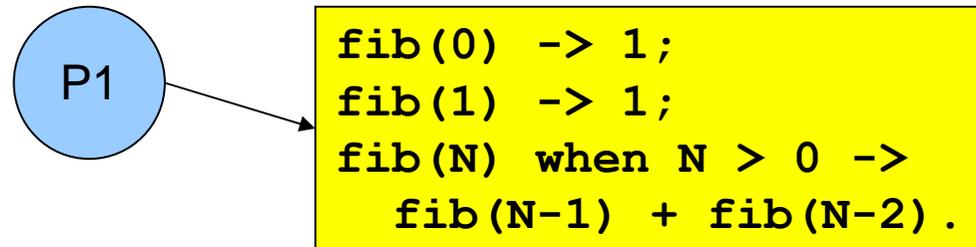
```
> dialyzer --add_to_plt --apps mnesia inets
```



Erlang: An Overview in Four Parts

Part 2 – Concurrency and Distribution

Thanks to Richard Carlsson for most of the slides in this part



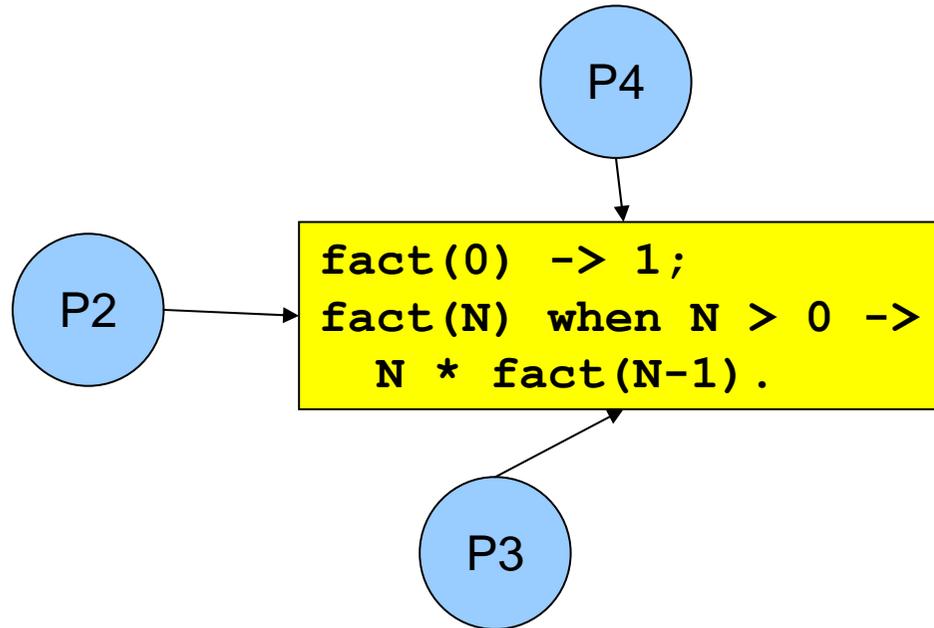
- Whenever an Erlang program is running, the code is executed by a *process*
- The process keeps track of the current program point, the values of variables, the call stack, etc.
- Each process has a unique *Process Identifier* (“*Pid*”), that can be used to identify the process
- *Processes are concurrent* (they can run in parallel)



Implementation

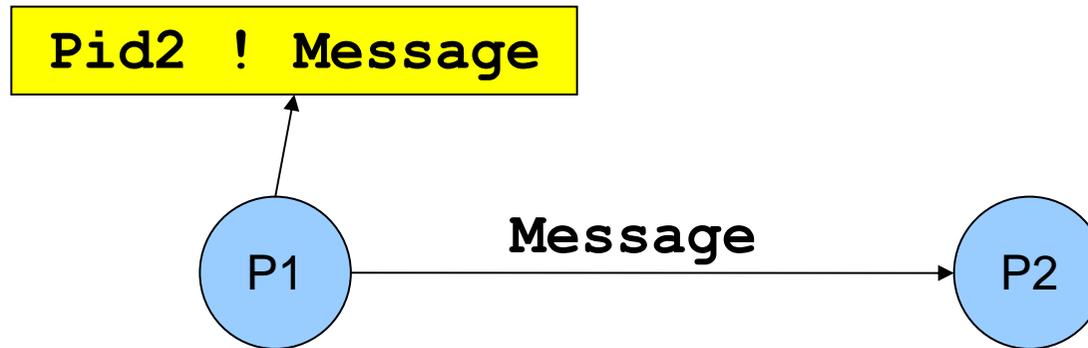
- Erlang processes are implemented by the VM's runtime system, not by operating system threads
- Multitasking is *preemptive* (the virtual machine does its own process switching and scheduling)
- Processes use very little memory, and switching between processes is very fast
- Erlang VM can handle large numbers of processes
 - Some applications use more than 100.000 processes
- On a multiprocessor/multicore machine, Erlang processes can be scheduled to run in parallel on separate CPUs/cores using multiple schedulers

Concurrent process execution



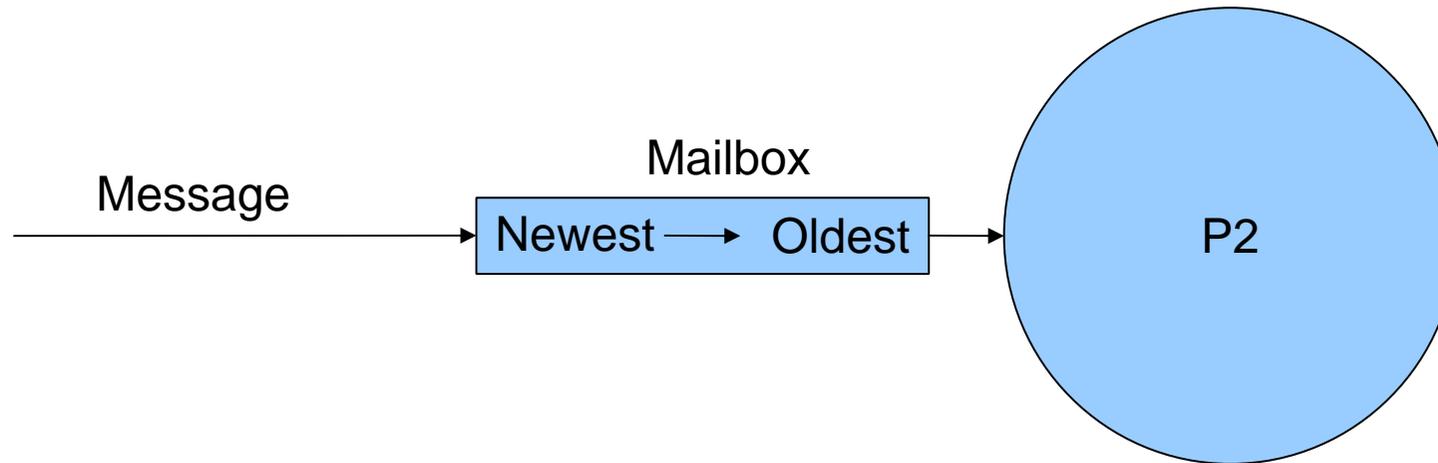
- Different processes may be reading the same program code at the same time
 - They have their own data, program point, and stack – only the text of the program is being shared (well, almost)
 - *The programmer does not have to think about other processes updating the variables*

Message passing



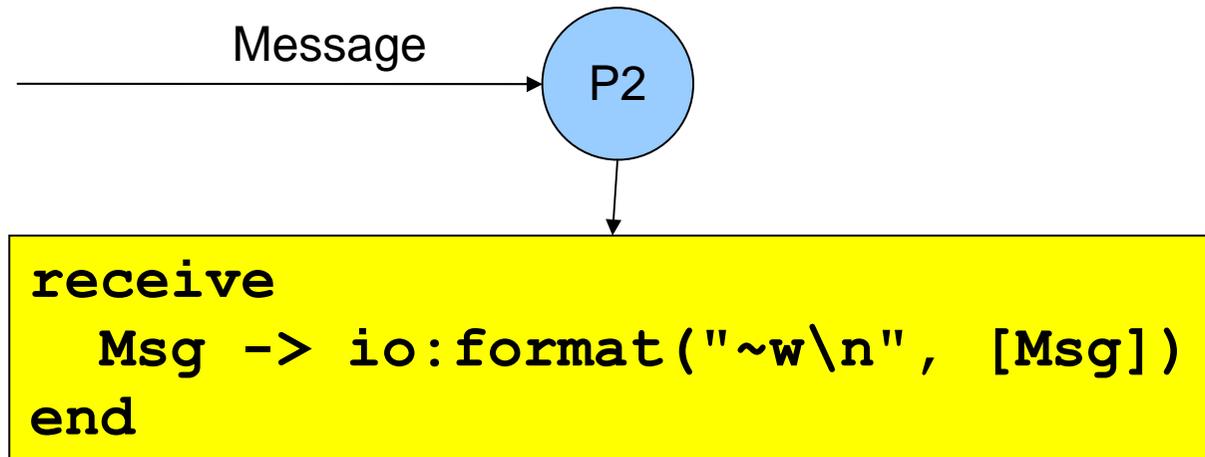
- “!” is the *send operator* (often called “bang!”)
 - The Pid of the receiver is used as the address
- Messages are sent *asynchronously*
 - The sender continues immediately
- Any value can be sent as a message

Message queues



- Each process has a *message queue* (mailbox)
 - Arriving messages are placed in the queue
 - *No size limit* – messages are kept until extracted
- A process *receives* a message when it extracts it from the mailbox
 - Does not have to take the first message in the queue

Receiving a message



receive expressions are similar to **case** switches

- Patterns are used to match messages in the mailbox
- Messages in the queue are tested in order
 - The first message that matches will be extracted
 - A variable-pattern will match the first message in the queue
- Only one message can be extracted each time

Selective receive

```
receive
  {foo, X, Y} -> ...;
  {bar, X} when ... -> ...;
  ...
end
```

- Patterns and guards let a programmer control the priority with which messages will be handled
 - Any other messages will remain in the mailbox
- The **receive** clauses are tried in order
 - If no clause matches, the next message is tried
- If *no* message in the mailbox matches, the process *suspends*, waiting for a new message

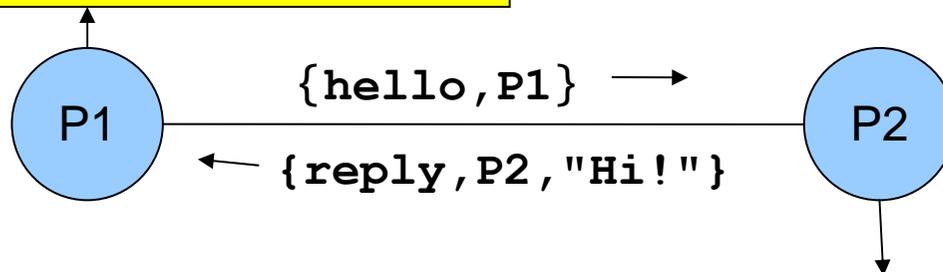
Receive with timeout

```
receive
  {foo, X, Y} -> ...;
  {bar, X} when ... -> ...
after 1000 ->
  ...           % handle timeout
end
```

- A **receive** expression can have an **after** part
 - The timeout value is either an integer (milliseconds), or the atom '**infinity**' (wait forever)
 - 0 (zero) means “just check the mailbox, then continue”
- The process will wait until a matching message arrives, or the timeout limit is exceeded
- **Soft real-time**: approximate, no strict timing guarantees

Send and reply

```
Pid ! {hello, self()},  
receive  
  {reply, Pid, String} ->  
    io:put_chars(String)  
end
```

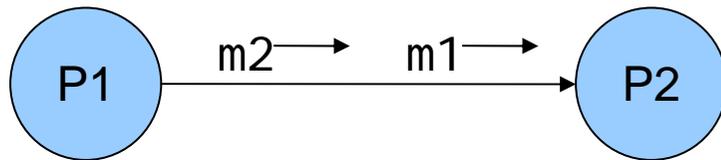


```
receive  
  {hello, Sender} ->  
    Sender ! {reply, self(), "Hi!"}  
end
```

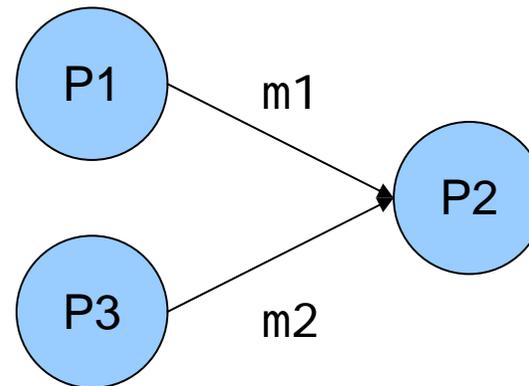
- Pids are often included in messages (`self()`), so the receiver can reply to the sender
 - If the reply includes the `Pid` of the second process, it is easier for the first process to recognize the reply

Message order

FIFO order
(same pair of sender and receiver)

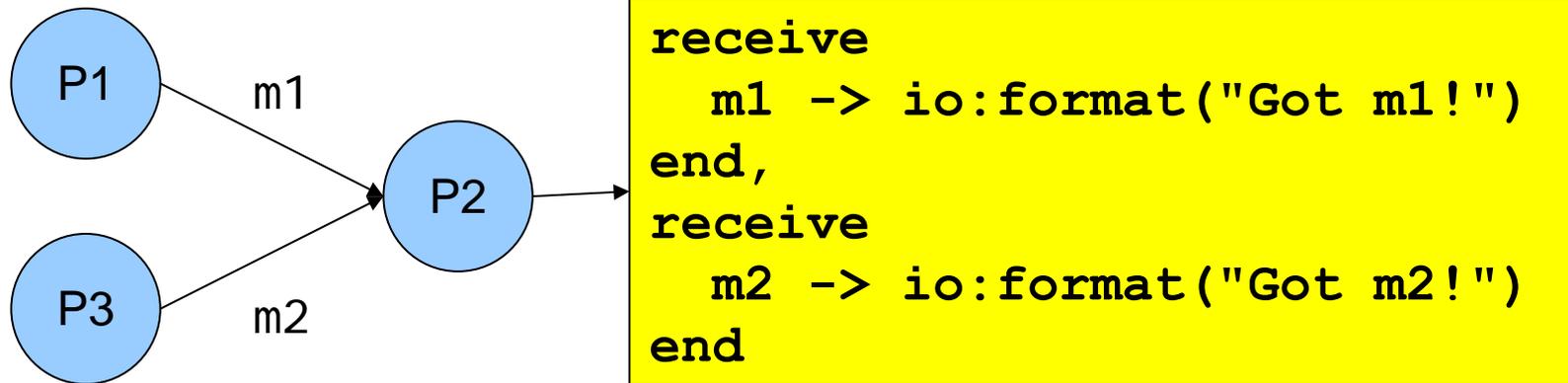


No guaranteed order
(different senders, same receiver)



- Within a node, the only guaranteed message order is when both the sender and receiver are the same for both messages (First-In, First-Out)
 - In the left figure, m1 will always arrive before m2 in the message queue of P2 (if m1 is sent before m2)
 - In the right figure, the arrival order can vary

Selecting unordered messages



- Using selective receive, we can choose which messages to accept, even if they arrive in a different order
- In this example, P2 will always print “Got m1!” before “Got m2!”, even if m2 arrives before m1
 - m2 will be ignored until m1 has been received

Starting processes

- The 'spawn' function creates a new process
- There are several versions of 'spawn':
 - `spawn(fun() -> ... end)`
 - can also do `spawn(fun f/0)` or `spawn(fun m:f/0)`
 - `spawn(Module, Function, [Arg1, ..., ArgN])`
 - `Module:Function/N` must be an exported function
- The new process will run the specified function
- The spawn operation always returns immediately
 - The return value is the Pid of the new process
 - The “parent” always knows the Pid of the “child”
 - The child will not know its parent unless you tell it

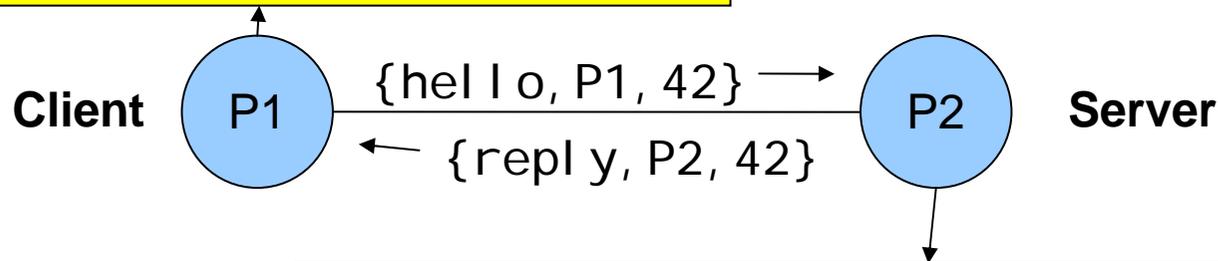
Process termination

- A process *terminates* when:
 - It finishes the function call that it started with
 - There is an exception that is not caught
 - The purpose of 'exit' exceptions is to terminate a process
 - “`exit(normal)`” is equivalent to finishing the initial call
- All messages sent to a terminated process will be thrown away, without any warning
 - No difference between throwing away and putting in mailbox just before process terminates
- The same process identifier will not be used again for a long time

A stateless server process

```
run() ->
  Pid = spawn(fun echo/0),

  Pid ! {hello, self(), 42},
  receive
    {reply, Pid, 42} ->
      Pid ! stop
  end.
```



```
echo() ->
  receive
    {hello, Sender, Value} ->
      Sender ! {reply, self(), Value},
      echo(); % loop!
  stop ->
    ok
  end.
```



A server process with state

```
server(State) ->
  receive
    {get, Sender} ->
      Sender ! {reply, self(), State},
      server(State);
    {set, Sender, Value} ->
      Sender ! {reply, self(), ok},
      server(Value);    % loop with new state!
  stop ->
    ok
  end.
```

- The parameter variables of a server loop can be used to remember the current state
- Note: the recursive calls to `server()` are *tail calls* (*last calls*) – *the loop does not use stack space*
- *A server like this can run forever*



A simple server example

```
-module(simple_server).  
-export([start/0]).  
  
-spec start() -> pid().  
start() ->  
    spawn(fun() -> loop(0) end).  
  
loop(Count) ->  
    NC = receive  
        {report, Pid} -> Pid ! Count;  
        _AnyOtherMsg -> Count + 1  
    end,  
    loop(NC).
```

```
Eshell V5.10.3 (abort ...^G)  
1> P = simple_server:start().  
<0.42.0>  
2> P ! foo.  
foo  
3> [P ! X || lists:seq(1,9)].  
[1,2,3,4,5,6,7,8,9]  
4> P ! {report, self()},  
    receive M -> M end.  
10
```



Hot code swapping

```
-module(server) .  
-export([start/0, loop/1]) .  
  
start() -> spawn(fun() -> loop(0) end) .  
  
loop(State) ->  
    receive  
        {get, Sender} ->  
            . . . ,  
            server:loop(State) ;  
        {set, Sender, Value} ->  
            . . . ,  
            server:loop(Value) ;  
        . . .
```

- When we use “`module:function(...)`”, Erlang will always call the latest version of the module
 - If we recompile and reload the `server` module, the process will jump to the new code after handling the next message – we can fix bugs without restarting!



Hiding message details

```
get_request(ServerPid) ->
    ServerPid ! {get, self()}.

set_request(Value, ServerPid) ->
    ServerPid ! {set, self(), Value}.

wait_for_reply(ServerPid) ->
    receive
        {reply, ServerPid, Value} -> Value
    end.

stop_server(ServerPid) ->
    ServerPid ! stop.
```

- Using interface functions keeps the clients from knowing about the format of the messages
 - You may need to change the message format later
- It is the client who calls the `self()` function here

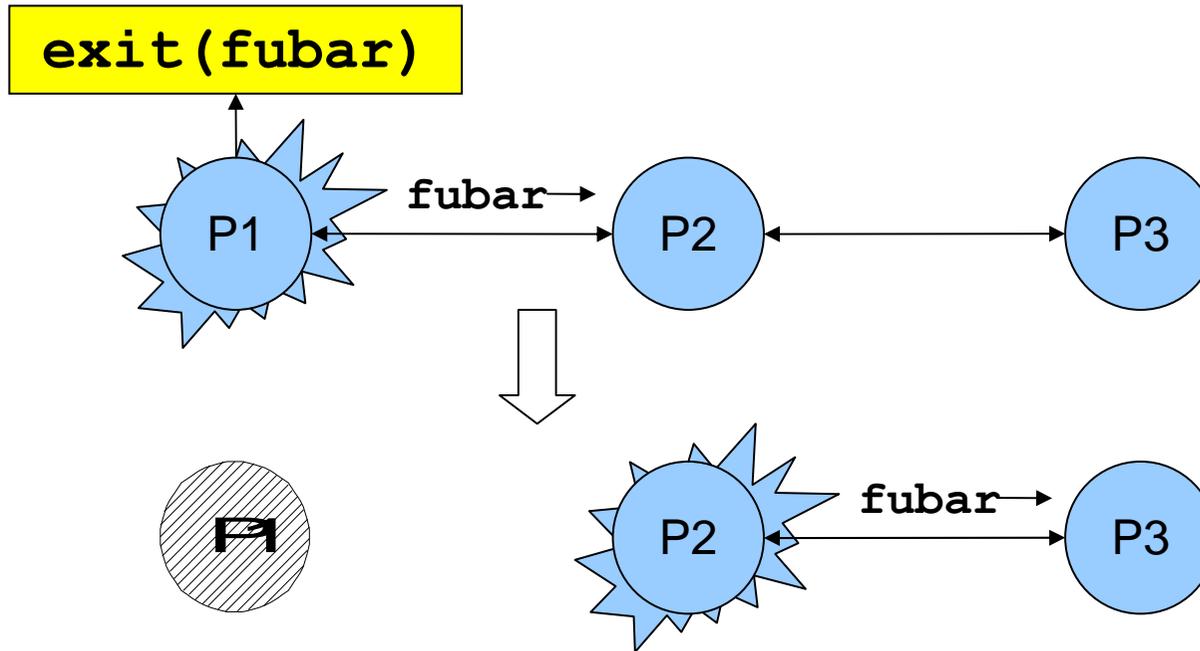


Registered processes

```
Pid = spawn(...),  
  
register(my_server, Pid),  
  
my_server ! {set, self(), 42},  
  
42 = get_request(my_server),  
  
Pid = whereis(my_server)
```

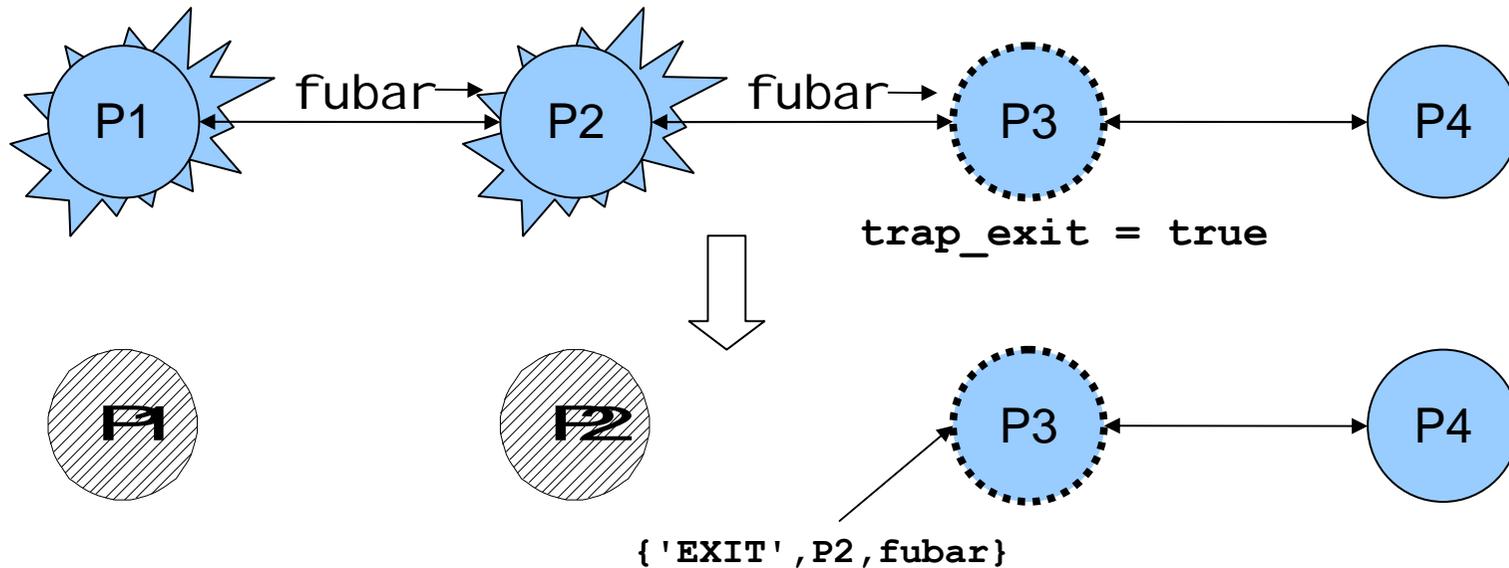
- A process can be registered under a name
 - the name can be any atom
- Any process can send a message to a registered process, or look up the Pid
- The Pid might change (if the process is restarted and re-registered), but the name stays the same

Links and exit signals



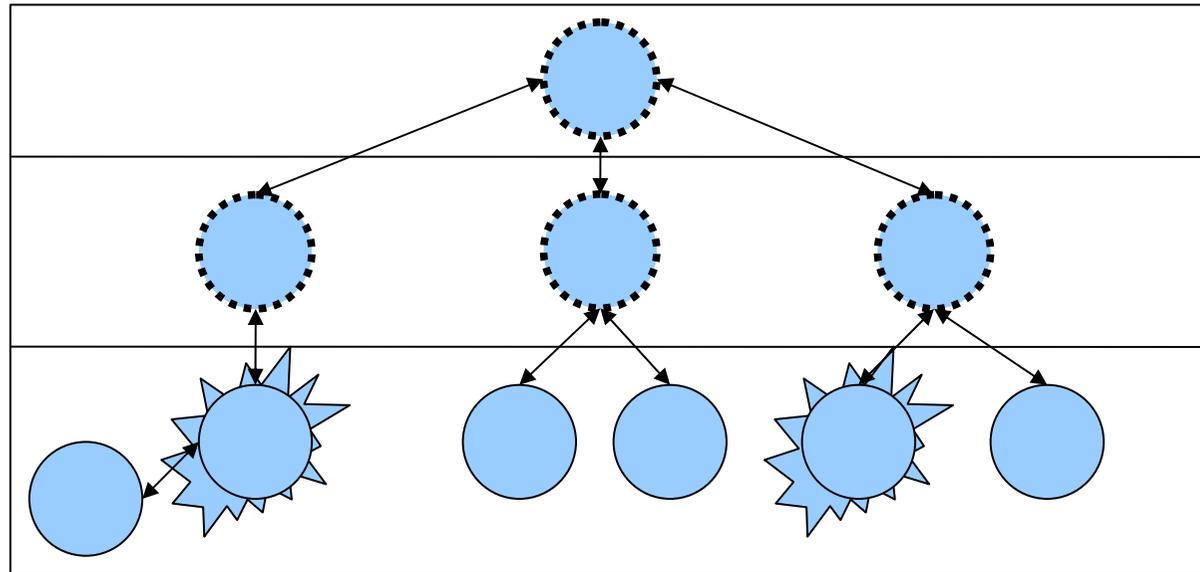
- Any two processes can be *linked*
 - Links are always bidirectional (two-way)
- When a process dies, an *exit signal* is sent to all linked processes, which are also killed
 - Normal exit does not kill other processes

Trapping exit signals



- If a process sets its `trap_exit` flag, all signals will be caught and turned into normal messages
 - `process_flag(trap_exit, true)`
 - `{'EXIT', Pid, ErrorTerm}`
- This way, a process can watch other processes
 - 2-way links guarantee that sub-processes are dead

Robust systems through layers



- Each layer supervises the next layer and restarts the processes if they crash
- The top layers use well-tested, very reliable libraries (OTP) that practically never crash
- The bottom layers may be complicated and less reliable programs that can crash or hang



Distribution

```
[foo.bar.se] $ erl -name fred
Erlang (BEAM) emulator version 5.10.3

Eshell V5.10.3 (abort with ^G)
(fred@foo.bar.se)1> node().
'fred@foo.bar.se'
(fred@foo.bar.se)2>
```

- Running “erl” with the flag “-name xxx”
 - starts the Erlang network distribution system
 - makes the virtual machine emulator a “node”
 - the node name is the atom 'xxx@host.domain'
- Erlang nodes can communicate over the network
 - but first they must find each other
 - simple security based on secret cookies

Connecting nodes

```
(fred@foo.bar.se) 2> net_adm:ping('barney@foo.bar.se').  
pong  
(fred@foo.bar.se) 3> net_adm:ping('wilma@foo.bar.se').  
pang  
(fred@foo.bar.se) 4>
```

- Nodes are connected the first time they try to communicate – after that, they stay in touch
 - A node can also supervise another node
- The function “`net_adm:ping(Node)`” is the easiest way to set up a connection between nodes
 - returns either “pong” or “pang” 😊
- We can also send a message to a registered process using “`{Name,Node} ! Message`”



Distribution is transparent

- One can send a Pid from one node to another
 - Pids are unique, even over different nodes
- We can send a message to *any* process through its Pid – even if the process is on another node
 - There is no difference (except that it takes more time to send messages over networks)
 - We don't have to know where processes are
 - We can make programs work on multiple computers with no changes at all in the code (no shared data)
- We can run several Erlang nodes (with different names) on the same computer – good for testing



Running remote processes

```
P = spawn('barney@foo.bar.se', Module, Function, ArgList),  
global:register_name(my_global_server, P),  
global:send(my_global_server, Message)
```

- We can use variants of the **spawn** function to start new processes directly on another node
- The module '**global**' contains functions for
 - registering and using named processes over the whole network of connected nodes
 - not same namespace as the local "register(...)"
 - must use "global:send(...)", not "!"
 - setting global locks



Ports – talking to the outside

```
PortId = open_port({spawn, "command"}, [binary]),  
PortId ! {self(), {command, Data}}  
  
PortId ! {self(), close}
```

- Talks to an external (or linked-in) C program
- A port is connected to the process that opened it
- The port sends data to the process in messages
 - binary object
 - packet (list of bytes)
 - one line at a time (list of bytes/characters)
- A process can send data to the port



Erlang: An Overview in Four Parts

Part 3 – OTP and Behaviours

OTP behaviours are a formalization of design patterns

- Processes have similar structures and life cycles
 - They are started
 - They receive messages and send replies
 - They are terminated (or crash)
- Even if they perform different tasks, they probably perform them following a set of patterns
- Each design pattern solves a specific problem



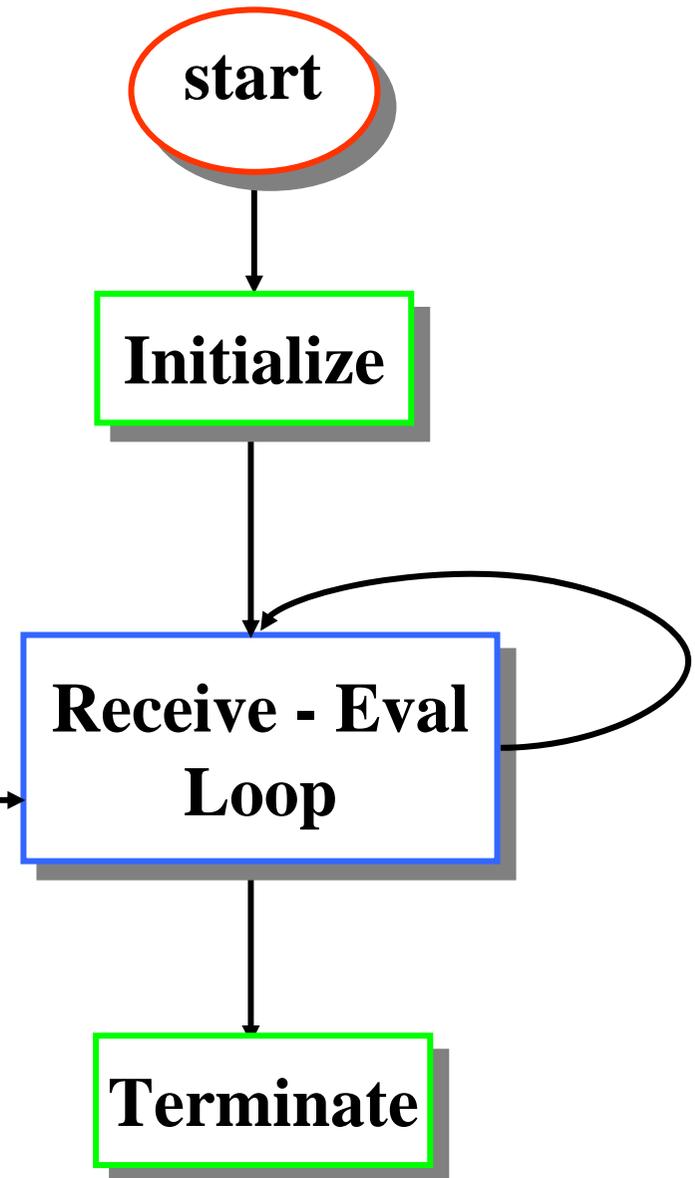
Design patterns

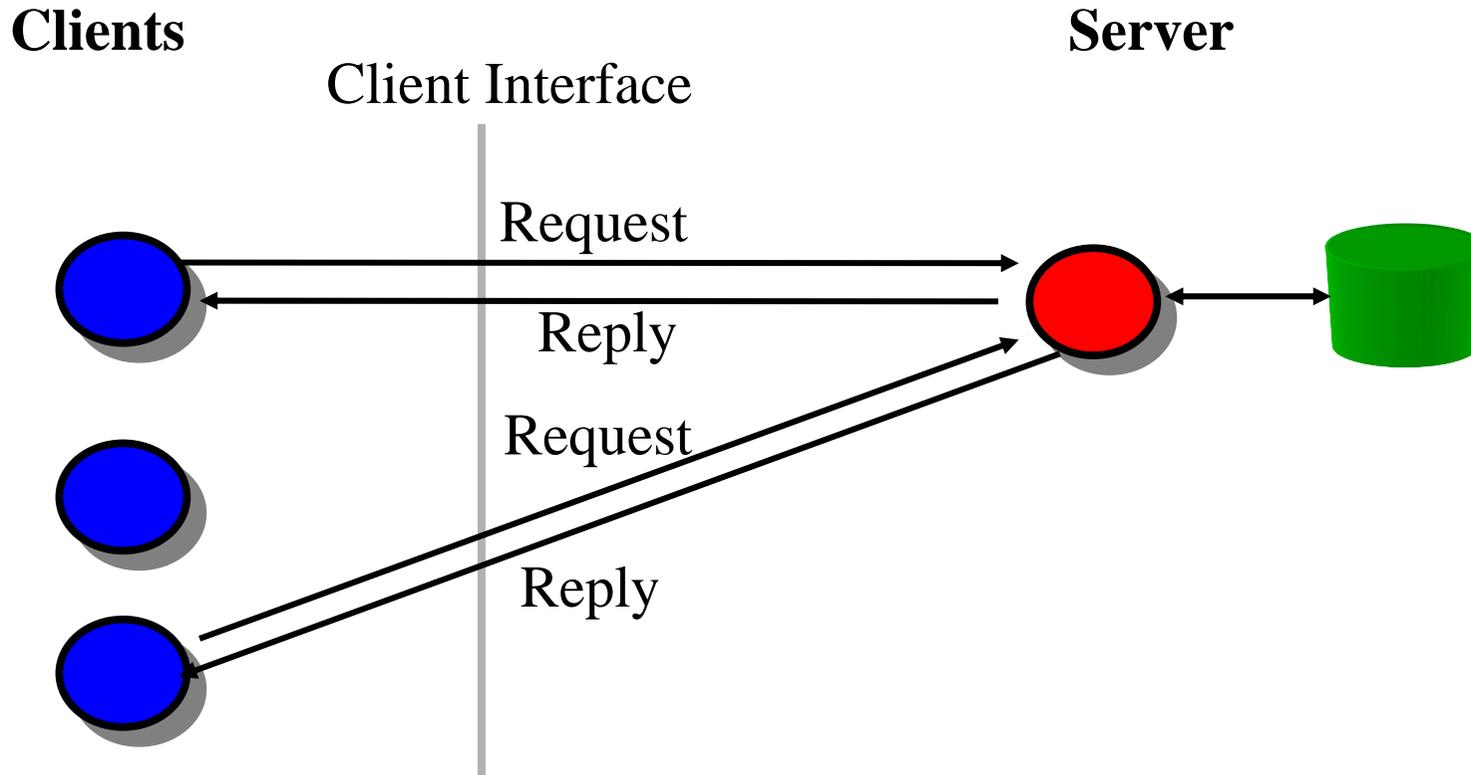
```
start(Args) ->  
  spawn(fun () -> init(Args) end).
```

```
init(Args) ->  
  State = initialize_state(Args),  
  loop(State).
```

```
loop(State) ->  
  receive  
    {handle, Msg} ->  
      NewState = handle(Msg, State),  
      loop(Mod, NewState);  
    stop ->    
      terminate(State)  
  end.
```

```
terminate(State) ->  
  cleanup(State).
```





Consider a Client - Server architecture

- What behaviours are different between systems?
- What similarities are there between systems?

Generic

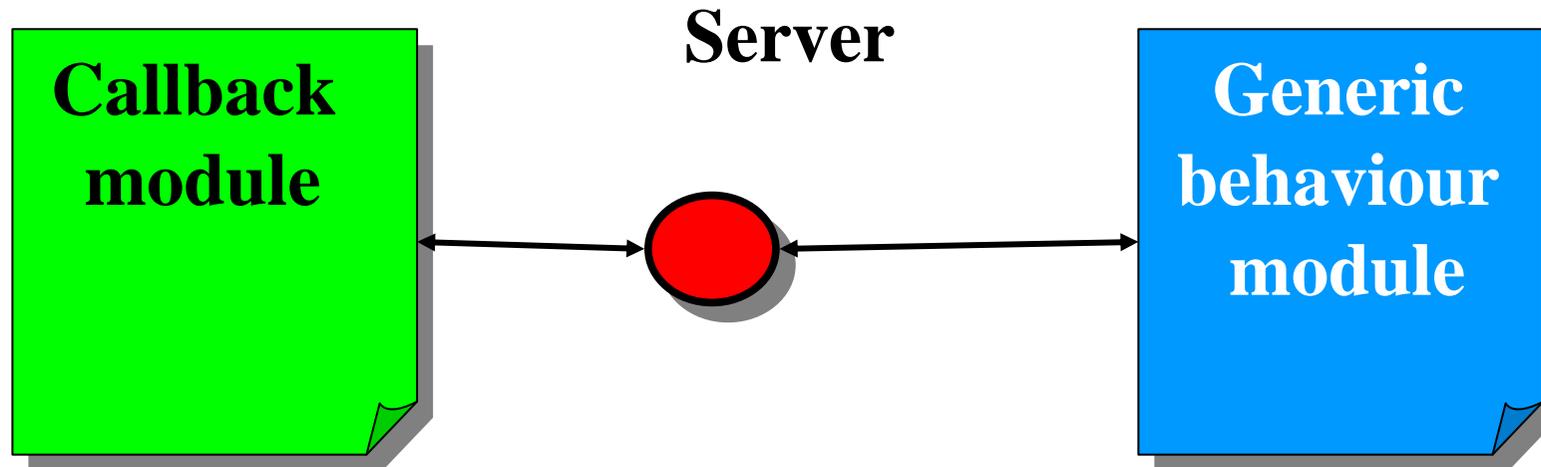
- Spawning the server
- Storing the loop data
- Sending requests to server
- Sending replies to client
- Receiving server replies
- Stopping the server

Specific

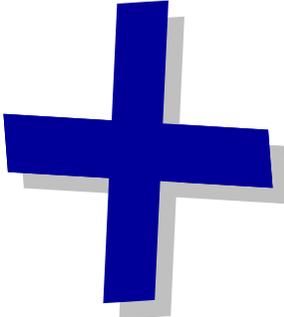
- Initialising the server state
- The loop data
- The client requests
- Handling client requests
- Contents of server reply
- Cleaning up

A behaviour 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 be passed to a generic module expecting the behaviour
- This can also be implemented rather easily using higher-order functions



- The idea is to split the code in two parts
- The generic part is called the generic behaviour
 - They are provided by OTP as library modules
- The specific part is called the callback module
 - They are implemented by the programmer



Pros

- Less code to develop
- Less bugs
- Solid well tested base
- Free built in functionality
 - Log, Trace, Statistics
 - Easy to add functionality
- Common programming style
- Component based terminology

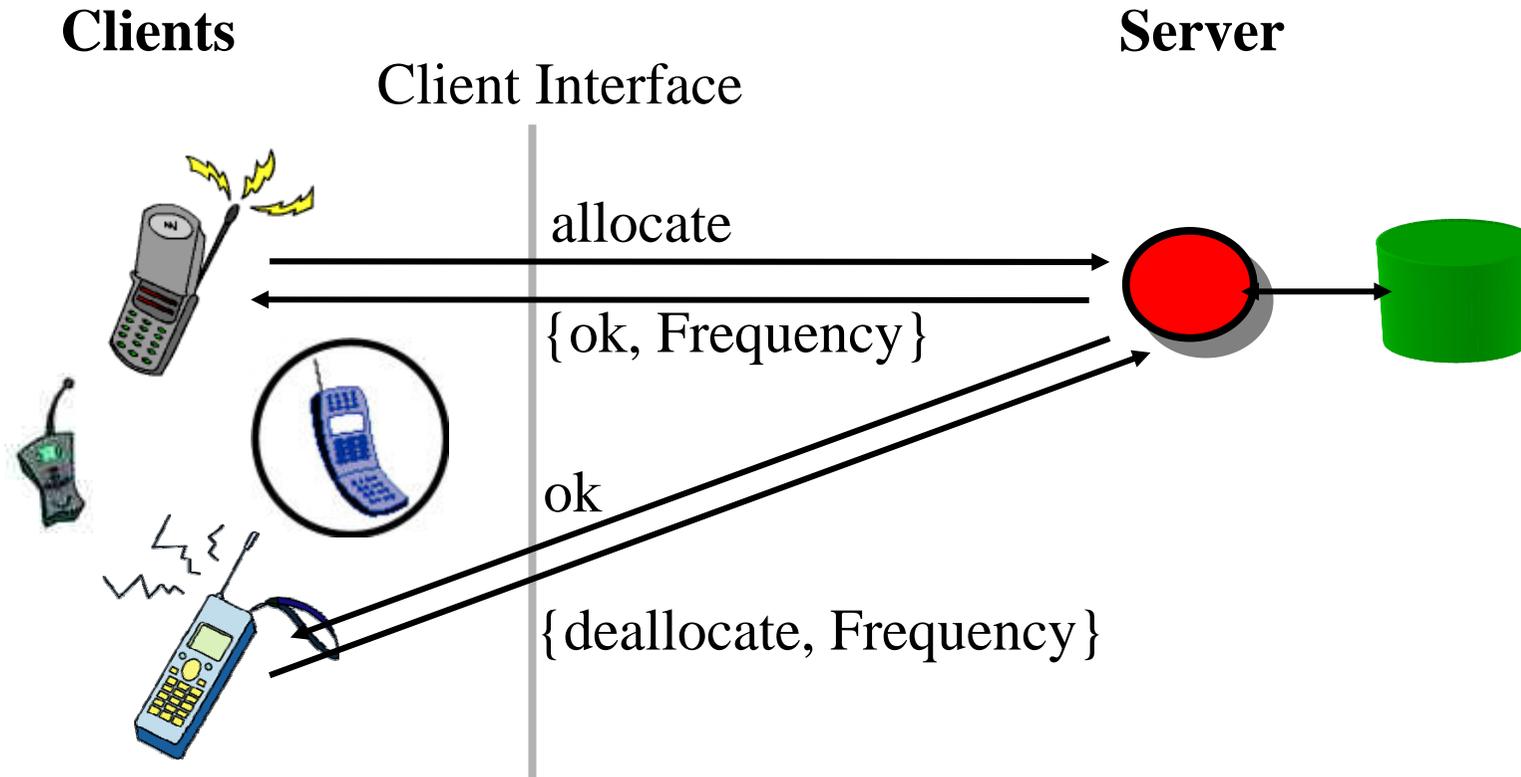


Cons

- Steep learning curve
- Affects performance

- Generic Server
 - Used to model client server behaviour
- Generic Finite State Machine
 - Used for finite state machine programming
- Generic Event Handler / Manager
 - Used for writing event handlers
- Supervisor
 - Used for fault-tolerant supervision trees
- Application
 - Used to encapsulate resources and functionality

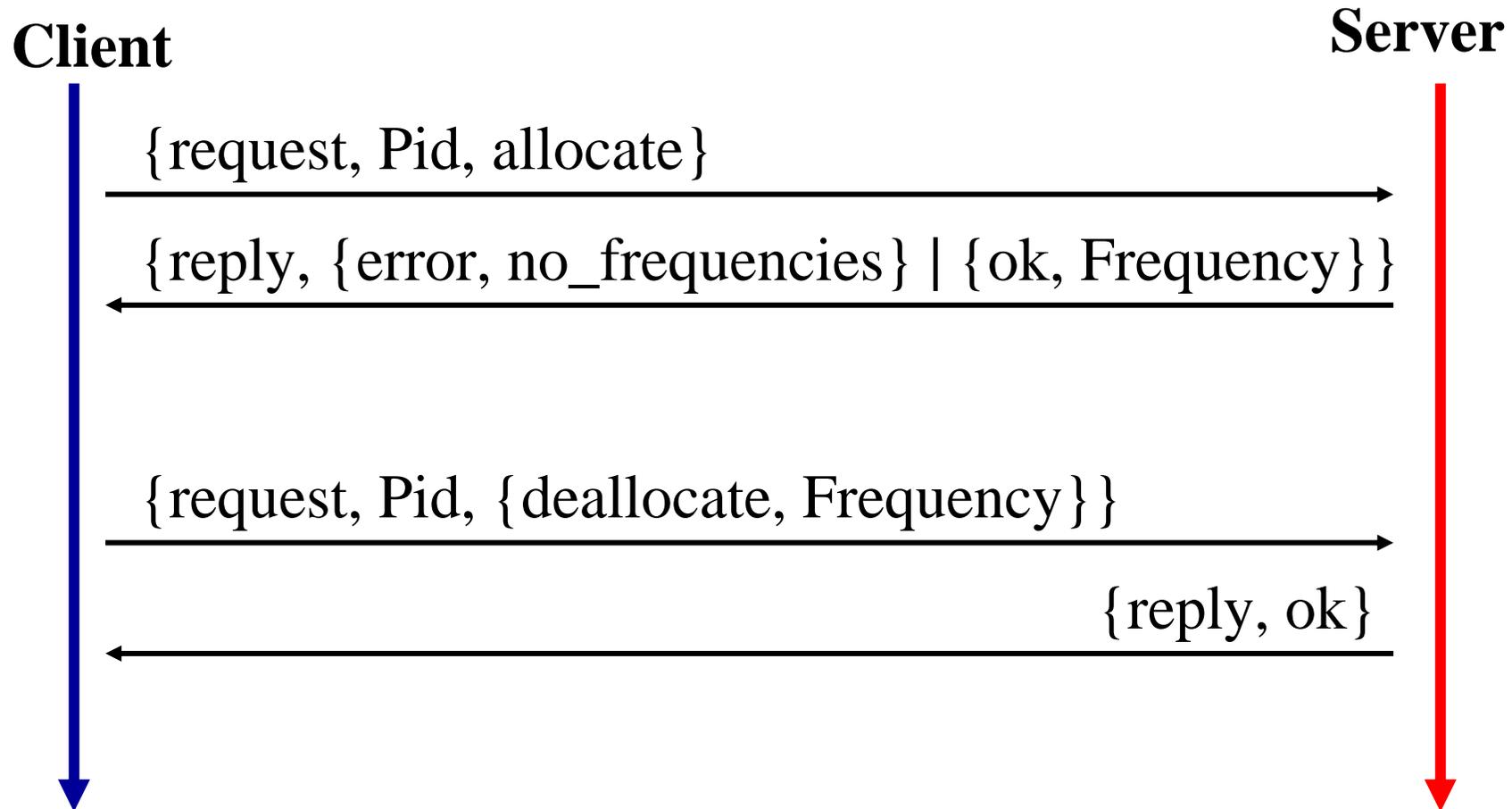
A server example



The following server is responsible for allocating and deallocating frequencies on behalf of mobile phones



A server example





A server example

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

```
%% These functions create and initialize the server
```

```
start() ->
```

```
    register(?MODULE, spawn(fun() -> init() end)).
```

```
init() ->
```

```
    Frequencies = {get_frequencies(), []},
```

```
    loop(Frequencies).
```

```
get_frequencies() -> lists:seq(17,42).
```

Generic



A server example

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

```
%% These functions create and initialize the server
```

```
start() ->
```

```
    register(?MODULE, spawn(fun () -> init() end)).
```

```
init() ->
```

```
    Frequencies = {get_frequencies(), []},
```

```
    loop(Frequencies).
```

```
get_frequencies() -> lists:seq(17,42).
```

Specific



A server example

%% 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.

Generic



A server example

%% 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.

Specific



A server example

%% 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.

Generic

reply(Pid, Message) ->
Pid ! {reply, Message}.



A server example

%% 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.
```

Specific

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



A server example

%% The Internal Functions

%% Help functions used to allocate and deallocate frequencies.

```
allocate({[], Allocated}, Pid) ->
```

```
  {{[], Allocated}, {error, no_frequencies}};
```

```
allocate({[Freq|Free], Allocated}, Pid) ->
```

```
  {{Free, [{Freq, Pid}|Allocated]}, {ok, Freq}}.
```

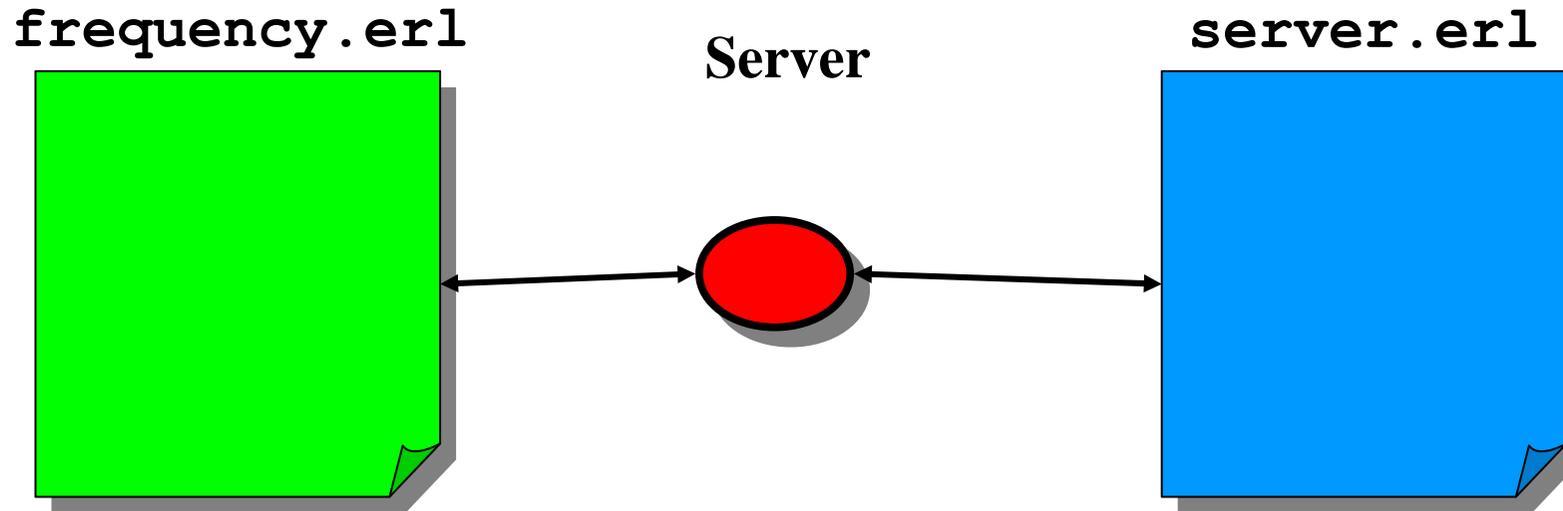
```
deallocate({Free, Allocated}, Freq) ->
```

```
  NewAllocated = lists:keydelete(Freq, 1, Allocated),
```

```
  {[Freq|Free], NewAllocated}.
```

Specific

A server example



- Place all the generic code in the module `server`
- Place all the specific code in the module `frequency`



A server example

server.erl

```
-module(server) .

-export([start/2, stop/1,
        call/2]).

start(Mod, Args) ->
    Fun = fun init(Mod, Args) end,
    Pid = spawn(Fun),
    register(Mod, Pid).

stop(Mod) ->
    Mod ! stop.

init(Mod, Args) ->
    State = Mod:init(Args),
    loop(Mod, State).
```

frequency.erl

```
-module(frequency) .

-export([start/0, stop/0,
        handle/2, allocate/0,
        init/1, deallocate/1]).

start() ->
    server:start(?MODULE, []).

stop() ->
    server:stop(?MODULE).

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

get_frequencies() ->
    lists:seq(17, 42).
```

Generic

Specific



A server example

server.erl

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

Generic

Specific

frequency.erl

```
allocate() ->
  server:call(?MODULE, {allocate, self()}).

deallocate(Freq) ->
  server:call(?MODULE, {deallocate, Freq}).
```



A server example

server.erl

```
loop(Mod, State) ->
  receive
    {request, Pid, Msg} ->
      {NewState, Reply} = Mod:handle(Msg, State),
      reply(Pid, Reply),
      loop(Mod, NewState);
    stop -> ok
  end.

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

Generic

frequency.erl

```
handle({allocate, Pid}, Frequencies) ->
  {NewFrequencies, Reply} = allocate(Frequencies, Pid),
  {NewFrequencies, Reply};

handle({deallocate, Freq}, Frequencies) ->
  Reply = ok,
  {deallocate(Frequencies, Freq), Reply}.
```

Specific



Erlang: An Overview in Four Parts

Part 4 – Testing Erlang Programs



A sorting program

```
%% my first sort program, inspired by QuickSort
-module(my_sort).
-export([sort/1]).

-spec sort([T]) -> [T].
sort([]) -> [];
sort([P|Xs]) ->
    sort([X || X <- Xs, X < P])
    ++ [P] ++ sort([X || X <- Xs, P < X]).
```

- How do we know that software works?
 - One popular method is to use testing
- Let's do manual testing of Erlang programs first
 - Relatively easy due to the interactive shell



Manual testing in the shell

```
Eshell V5.10.3 (abort with ^G)
1> c(my_sort) .
{ok,my_sort}
2> my_sort:sort([]) .
[]
3> my_sort:sort([17,42]) .
[17,42]
4> my_sort:sort([42,17]) .
[17,42]
5> my_sort:sort([3,1,2]) .
[1,2,3]
```

- Seems to work!
- However, perhaps it's not a good idea to execute these tests repeatedly by hand
 - Let's put them in the file...
 - ... and exploit the power of pattern matching



A sorting program with unit tests

```
-module(my_sort).  
-export([sort/1, sort_test/0]).  
  
-spec sort([T]) -> [T].  
sort([]) -> [];  
sort([P|Xs]) ->  
    sort([X || X <- Xs, X < P])  
    ++ [P] ++ sort([X || X <- Xs, P < X]).  
  
-spec sort_test() -> ok.  
sort_test() ->  
    [] = sort([]),  
    [17,42] = sort([17,42]),  
    [17,42] = sort([42,17]),  
    [1,2,3] = sort([3,1,2]),  
    ok.
```

Convention:
program code in this and
the following slides use
boldface for showing the
parts of the program that
were added or changed
w.r.t. the previous code

And now let's use EUnit to run them automatically



Running tests using EUnit

```
6> my_sort:sort_test().  
ok  
7> eunit:test(my_sort).  
Test passed.  
ok
```

- EUnit in its simplest form is a test framework to automatically run all `_test` functions in a module
- Calling `eunit:test(Module)` was all that was needed here
- However, EUnit can do much more...

Let us, temporarily, change one test to:

```
[1,3,2] = sort([3,1,2])
```

and see what happens



EUnit and failures

```
8> c(my_sort) .
{ok,my_sort}
9> eunit:test(my_sort) .
my_sort: sort_test (module 'my_sort')...*failed*
in function my_sort:sort_test/0 (my_sort.erl, line 13)
** error:{badmatch,[1,2,3]}

=====
Failed: 1.   Skipped: 0.   Passed: 0.
error
```

- Reports number of tests that failed and why
 - the report is pretty good, but it can get even better
 - using EUnit macros



A sorting program with EUnit tests

```
%% my first sort program, inspired by QuickSort
-module(my_sort).
-export([sort/1, sort_test/0]).

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

-spec sort([T]) -> [T].
sort([]) -> [];
sort([P|Xs]) ->
    sort([X || X <- Xs, X < P])
    ++ [P] ++ sort([X || X <- Xs, P < X]).

-spec sort_test() -> ok.
sort_test() ->
    ?assertEqual([], sort([])),
    ?assertEqual([17,42], sort([17,42])),
    ?assertEqual([17,42], sort([42,17])),
    ?assertEqual([1,3,2], sort([3,1,2])),
    ok.
```



Unit testing using EUnit macros

```
10> c(my_sort) .
my_sort.erl:2 Warning: function sort_test/0 already exported
{ok,my_sort}
11> eunit:test(my_sort) .
my_sort: sort_test (module 'my_sort')...*failed*
in function my_sort:'-sort_test/0-fun...'/1 (my_sort.erl, line 15)
in call from my_sort:sort_test/0 (my_sort.erl, line 15)
** error:{assertEqual_failed, [{module,my_sort},
                               {line,15},
                               {expression,"sort ( [3,1,2] )"},
                               {expected,[1,3,2]},
                               {value,[1,2,3]}}]
=====
Failed: 1. Skipped: 0. Passed: 0.
error
```

- This report is much more detailed
- But, it considers the complete set of tests as one



EUnit test generators

```
-module(my_sort).  
-export([sort/1]).  
  
-include_lib("eunit/include/eunit.hrl").  
  
sort([]) -> ...  
  
sort_test_() ->    % notice trailing underscore  
    [test_zero(), test_two(), test_three()].  
  
test_zero() ->  
    [?_assertEqual([], sort([]))]. % notice underscores  
test_two() ->  
    [?_assertEqual([17,42], sort([17,42])),  
     ?_assertEqual([17,42], sort([42,17]))].  
test_three() ->  
    [?_assertEqual([1,3,2], sort([3,1,2]))].
```



EUnit test generators

```
12> c(my_sort) .
{ok,my_sort}
13> eunit:test(my_sort) .
my_sort:20 test_three...*failed*
in function my_sort:'-test_three/0-fun...'/1 (my_sort.erl, line 20)
** error:{assertEqual_failed,[{module,my_sort},
                               {line,20},
                               {expression,"sort ( [3,1,2] )"},
                               {expected,[1,3,2]},
                               {value,[1,2,3]}}}]
=====
Failed: 1. Skipped: 0. Passed: 3.
error
```

- EUnit now reports accurate numbers of passed and failed test cases
- In fact, we can test EUnit generators individually



EUnit test generators

```
14> eunit:test({generator, fun my_sort:sort_test_/0}).
my_sort:20 test_three...*failed*
in function my_sort:'-test_three/0-fun...'/1 (my_sort.erl, line 20)
** error:{assertEqual_failed, [{module,my_sort},
                               {line,20},
                               {expression,"sort ( [3,1,2] )"},
                               {expected,[1,3,2]},
                               {value,[1,2,3]}}

=====
Failed: 1. Skipped: 0. Passed: 3.
error
```

- This works only for test generator functions
(not very impressive, as there is only one in this example)
- There are other forms that may come handy (RTFM)
e.g. `{dir, Path}` to run all tests for the modules in `Path`



EUnit test generators

- Let us undo the error in the `test_three` test
- add one more EUnit generator

```
another_sort_test_() ->
    [test_four()].

test_four() ->
    [?_assertEqual([1,2,3,4], sort([1,3,2,4])),
     ?_assertEqual([1,2,3,4], sort([1,4,2,3]))].
```

- and run again: all tests and just the new ones

```
15> c(my_sort).
{ok,my_sort}
16> eunit:test(my_sort).
    All 6 tests passed
ok
17> eunit:test({generator, fun my_sort:another_sort_test_/0}).
    All 2 tests passed
ok
```



There is more to EUnit...

- More macros
 - Utility, assert, debugging, controlling compilation
- Support to run tests in parallel
- Lazy generators
- *Fixtures* for adding scaffolding around tests
 - Allow to define setup and teardown functions for the state that each of the tests may need
 - Useful for testing stateful systems

For more information consult the EUnit manual

Towards automated testing

- Testing accounts for a large part of software cost
- Writing (unit) tests by hand is
 - boring and tedious
 - difficult to be convinced that all cases were covered
- Why not automate the process?
 - Yes, but how?
- One approach is **property-based testing**
 - Instead of writing test cases, let's write properties that we would like our software (functions) to satisfy
 - and use a tool that can automatically generate random inputs to test these properties



Property for the sorting program

```
-module(my_sort).
-export([sort/1]).

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

-spec sort([T]) -> [T].
sort([]) -> [];
sort([P|Xs]) ->
    sort([X || X <- Xs, X < P])
    ++ [P] ++ sort([X || X <- Xs, P < X]).

prop_ordered() ->
    ?FORALL(L, list(integer()), ordered(sort(L))).

ordered([]) -> true;
ordered([_]) -> true;
ordered([A,B|T]) -> A =< B andalso ordered([B|T]).
```



Testing the ordered property

```
$ erl -pa /path/to/proper/ebin
Erlang (BEAM) emulator version 5.10.3

Eshell V5.10.3 (abort with ^G)
1> c(my_sort).
{ok,my_sort}
2> proper:quickcheck(my_sort:prop_ordered()).
..... 100 dots .....
OK: Passed 100 tests
true
3> proper:quickcheck(my_sort:prop_ordered(), 10000).
..... 10000 dots .....
OK: Passed 10000 tests
true
```

- Runs any number of “random” tests we feel like
- If all tests satisfy the property, reports that all test passed



Another property for sorting

```
-module(my_sort).
-export([sort/1]).

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

-spec sort([T]) -> [T].
sort([]) -> [];
sort([P|Xs]) ->
    sort([X || X <- Xs, X < P])
    ++ [P] ++ sort([X || X <- Xs, P < X]).

prop_ordered() ->
    ?FORALL(L, list(integer()), ordered(sort(L))).

prop_same_length() ->
    ?FORALL(L, list(integer()),
        length(L) == length(sort(L))).

ordered([]) -> ...
```



Testing the same length property

```
4> c(my_sort) .
{ok,my_sort}
5> proper:quickcheck(my_sort:prop_same_length()) .
.....!
Failed: After 6 test(s) .
[0,0]

Shrinking (0 time(s))
[0,0]
false
6> proper:quickcheck(my_sort:prop_same_length()) .
.....!
Failed: After 13 test(s) .
[2,-8,-3,1,1]

Shrinking .(1 time(s))
[1,1]
false
```



Properties with preconditions

- Let us suppose that we actually *wanted* that our program only sorts for lists without duplicates
- How would we have to write the property then?

```
prop_same_length() ->
  ?FORALL(L, list(integer()),
    ?IMPLIES(no_duplicates(L),
      length(L) == length(sort(L)))).

%% better implementations of no_duplicates/1 exist
no_duplicates([]) -> true;
no_duplicates([A|T]) ->
  not lists:member(A, T) andalso no_duplicates(T).
```

```
7> proper:quickcheck(my_sort:prop_same_length()).
.....x.x.....x.xx.x...xx.xxxx.....x...xx.xxx
.....xx.x.x.....x.x.x.x.x.....xxxxx.xxxxxx...x.x.x.x.x.
OK: Passed 100 tests
```

Custom generators

- An even better way is to try to generate lists without duplicates in the first place!

```
list_no_dupls(T) ->
    ?LET(L, list(T), remove_duplicates(L)).

%% better versions of remove_duplicates/1 exist
remove_duplicates([]) -> [];
remove_duplicates([A|T]) ->
    case lists:member(A, T) of
        true -> remove_duplicates(T);
        false -> [A|remove_duplicates(T)]
    end.
```

```
prop_same_length() ->
    ?FORALL(L, list_no_dupls(integer()),
            length(L) == length(sort(L))).
```

```
7> proper:quickcheck(my_sort:prop_same_length()).
..... 100 dots .....
OK: Passed 100 tests
```



Testing for stronger properties

- Ok, but the properties we tested were quite weak
- How about ensuring that the list after sorting has the same elements as the original one?
- We can use some `obviously correct' function as reference implementation and test equivalence

```
prop_equiv_usort() ->
  ?FORALL(L, list(integer()),
    sort(L) ::= lists:usort(L)).
```

```
8> proper:quickcheck(my_sort:prop_equiv_usort()).
..... 100 dots .....
OK: Passed 100 tests
```

Note: PropEr is ideally suited for easily checking equivalence of two functions and gradually refining or optimizing one of them!



Beyond monotypic testing

- But why were we testing for lists of integers?
- We do not have to! We can test for general lists!

```
prop_equiv_usort() ->  
  ?FORALL(L, list(), sort(L) == lists:usort(L)).
```

```
9> proper:quickcheck(my_sort:prop_equiv_usort()).  
..... 100 dots .....  
OK: Passed 100 tests
```



Shrinking general terms

- How does shrinking work in this case?
- Let's modify the property to a false one and see

```
prop_equiv_sort() ->  
  ?FORALL(L, list(), sort(L) == lists:sort(L)).
```

```
10> proper:quickcheck(my_sort:prop_equiv_sort()).  
.....!  
Failed: After 14 test(s)  
[[[]], [ <<54,17,42:7>> ], 4], {}, -0.05423250622902363, {}, {42, <<0:3>>}]  
  
Shrinking ... (3 time(s))  
[[], {}]  
false  
11> proper:quickcheck(my_sort:prop_equiv_sort()).  
.....!  
Failed: After 28 test(s)  
[[], {[], 6, 'f%Co', {42}, ..... A REALLY BIG COMPLICATED TERM HERE  
CONTAINING TWO EMPTY LISTS  
  
Shrinking .... (4 time(s))  
[[[]], []]  
false
```

Testing frameworks

	Unit Testing	Property-Based Testing
Acquire a valid input	User-provided inputs	Generated semi-randomly from specification
Run the program	Automatic	Automatic
Decide if it passes	User-provided expected outputs	Partial correctness property