Lecture 9

Erlang

Erlang

- Functional
- Concurrent
- Distributed
- “Soft” real-time
- OTP (fault-tolerance, hot code update…)
- Open
  - Check the source code of generic behaviours

Functional

- Haskell – call-by-need (lazy)
  - Expression are evaluated ”just-in-time”
  - Hard to predict memory and execution behaviour
- Erlang – call-by-value (strict)
  - Combinator style OK
  - Direct pattern matching → tail recursion
  - Process → tail recursion
    - Constant memory usage

Recursion

```
-module(list_stuff).
-export([append/2, reverse/1]).

append([X|Xs], Ys) ->
   [X | append(Xs, Ys)];
average([],    Ys) ->
   Ys.

reverse([H|T]) ->
   append(reverse(T), [H]);
reverse([])    ->
   [].
```

Tail Recursion

```
-module(list_stuff).
-export([reverse/1, append/2]).

reverse(Xs) -> reverse_a(Xs, []).

reverse_a([X|Xs], Acc) ->
    reverse_a(Xs, [X|Acc]);
reverse_a([],     Acc) ->
    Acc.

%%continues
```

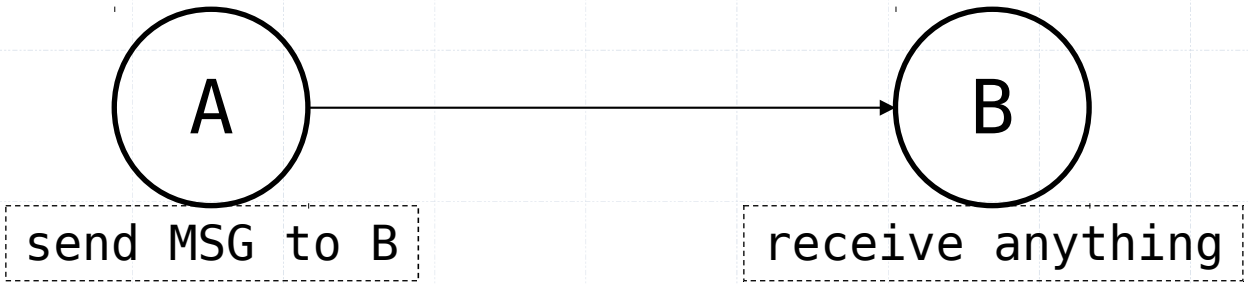
Tail Recursion

```
%%continuation

append(Xs, Ys) ->
   reverse_a(reverse(Xs),Ys).
```

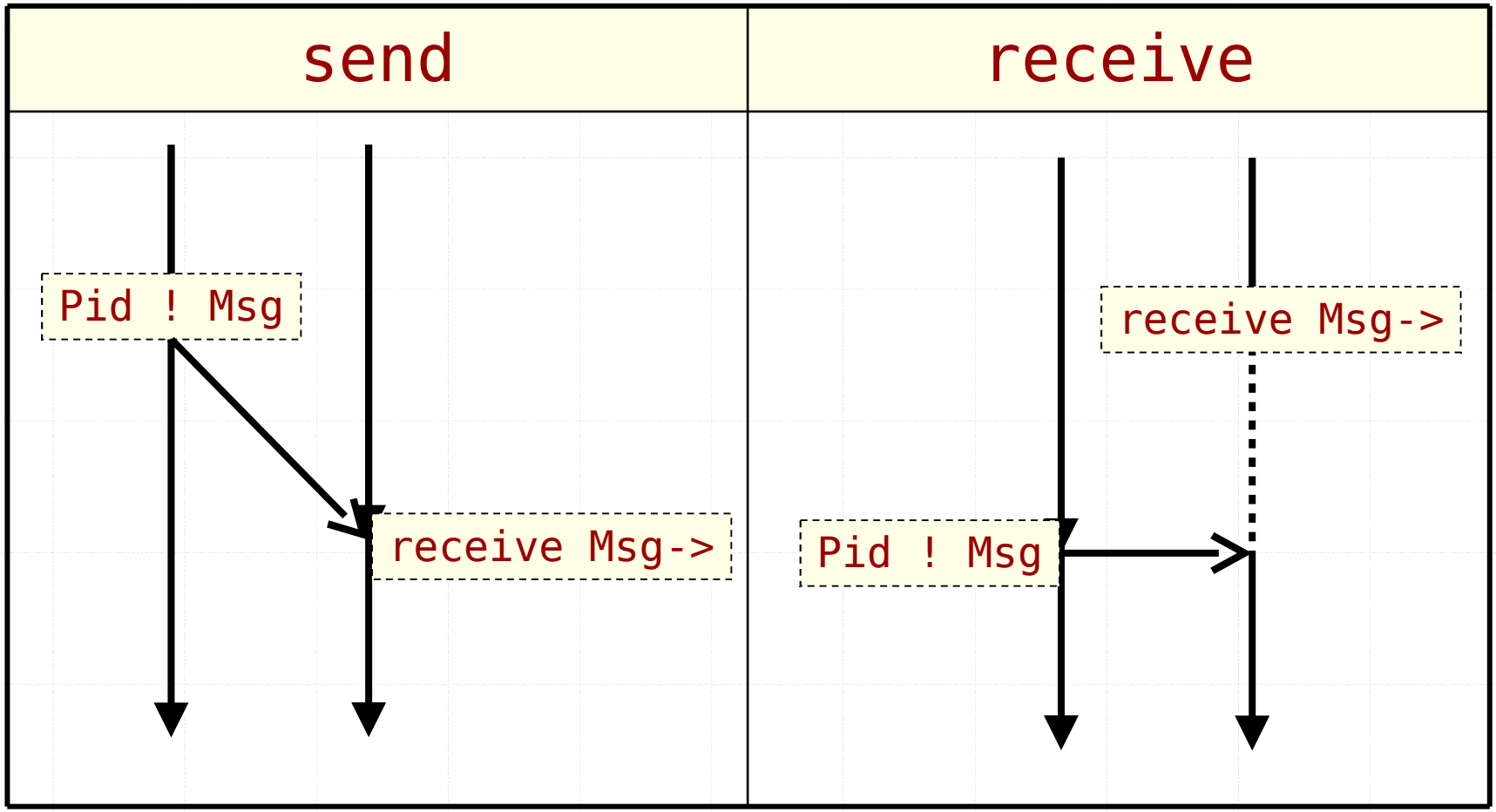
Concurrent Programming

- Based on Message Passing:
  - Q. What form of synchronisation?
  - A. Asynchronous
  - Q. What form of process naming?
  - A. Direct, asymmetric



Asynchronous Message Passing

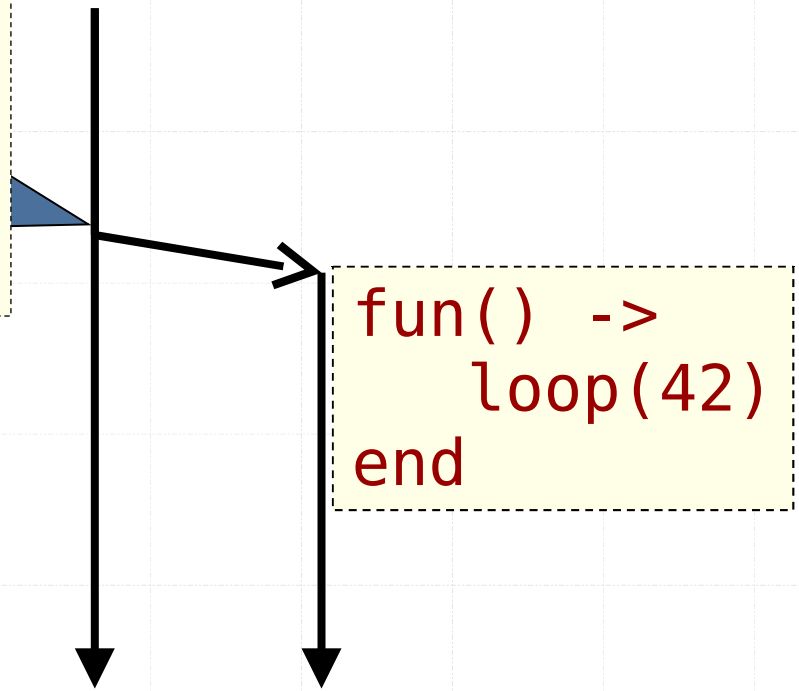
- Asynchronous send, receive from mailbox



Concurrent

- Spawn a function evaluation in a separate thread

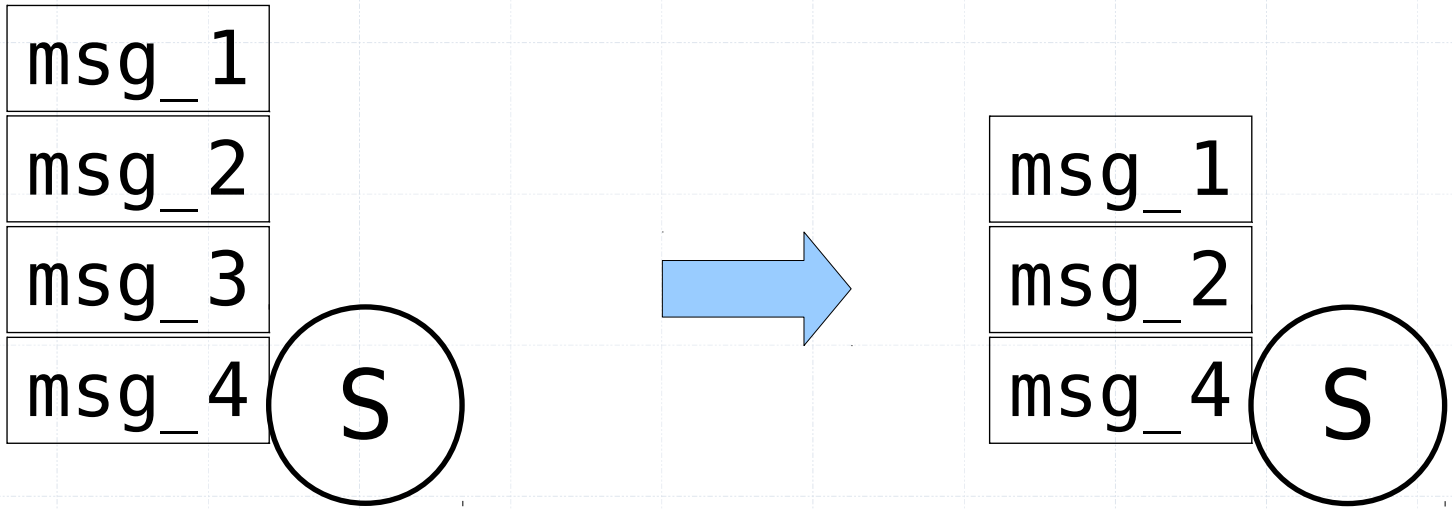
```
Pid = spawn(
          fun() ->
             loop(42)
          end)
```



- Function must use constant space
  - Tail recursive

Receive Message Order

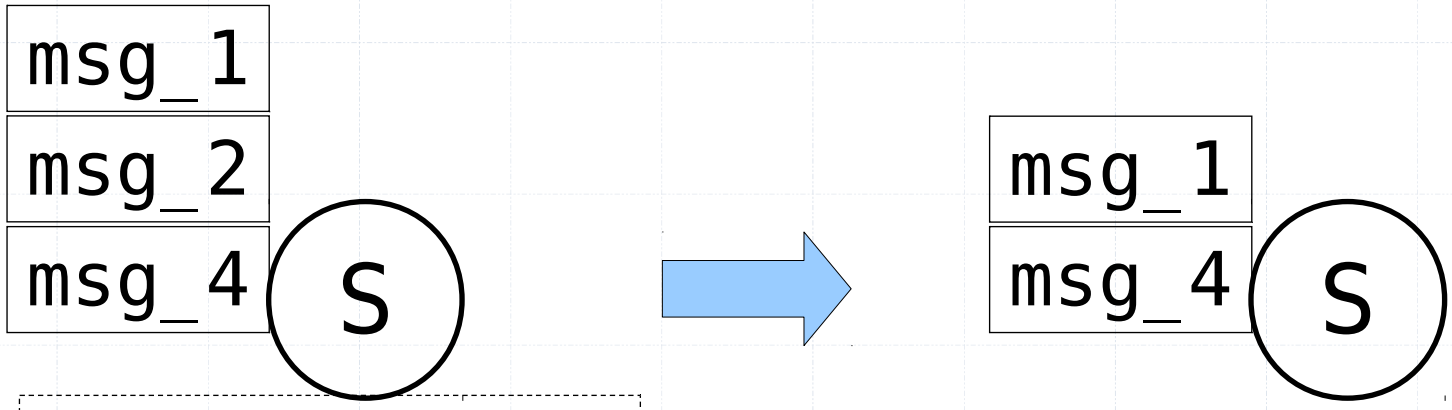
- A receive statement tries to find a match as early in the mailbox as it can




```
receive
    msg_3 -> …
end
```

Receive Message Order

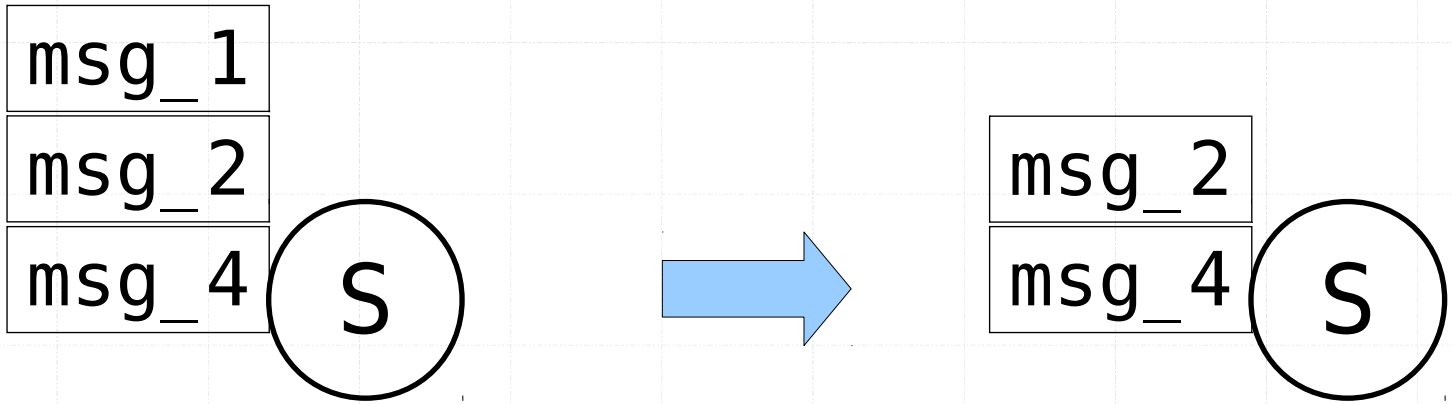
- A receive statement tries to find a match as early in the mailbox as it can




```
receive
    msg_4 -> …
    msg_2 -> …
end
```

Receive Message Order

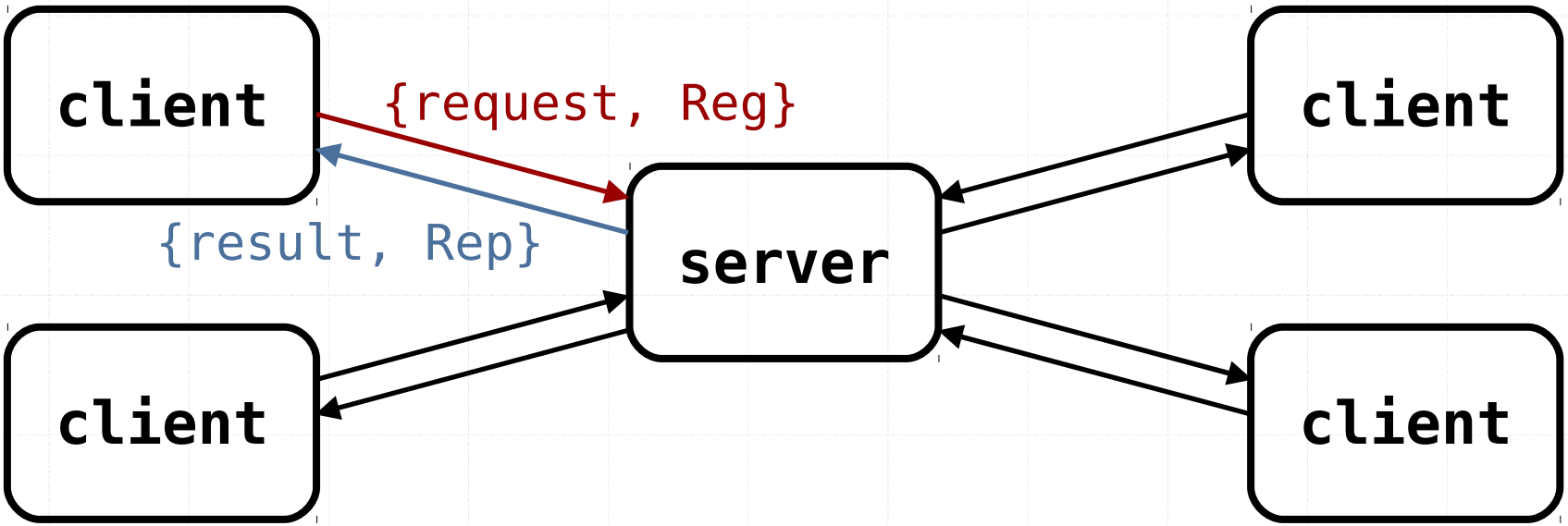
- A receive statement tries to find a match as early in the mailbox as it can




```
receive
    msg_4 -> …
    _  -> …
end
```

Client-Server Interaction

- Common asynchronous communication pattern
  - For example: a web server handles requests for web pages from clients (web browsers)



Modelling Client-Server

- Computational server
  - Off-loading heavy mathematical operations
  - For example: factorial

```
-module(g_server).
-import(math_stuff,[factorial/1]).
-export([start/0]).

start() -> spawn(fun() -> loop(0) end).

%%continues
```

Main Server Loop

```
%%continuation
loop(Count) ->
   receive
      {factorial, From, N} ->
         Result = factorial(N),
         From ! {result, Result},
         loop(Count+1);
      {get_count, From} ->
         From ! {result, Count},
         loop(Count);
      stop -> true
   end.
```

Client Interface

- Encapsulating client access in a function
- Private channel for receiving replies?

```
%%possible continuation

compute_factorial(Pid, N) ->
   Pid!{factorial, self(), N},
   receive
      {result, Result} ->
         Result
   end.
```

References

- Private channel for receiving replies?
  - Find the corresponding reply in the mailbox

```
receive
   {result, Result} -> Result
end
```

- BIF `make_ref()`
  - Provides globally unique object different from every other object in the Erlang system including remote nodes

RPC – Client Interface

- References to uniquely identify messages
  - References allow only matching

```
%%usual continuation
compute_factorial(Pid, N) ->
    Ref = make_ref(),
    Pid!{factorial, self(), Ref, N},
    receive
        {result, Ref, Result} ->
            Result
    end.
```

RPC – Main Server Loop

```
%%continuation
loop(Count) ->
   receive
      {factorial, From, Ref, N} ->
         Result = factorial(N),
         From ! {result, Ref, Result},
         loop(Count+1);
      {get_count, From, Ref} ->
         From ! {result, Ref, Count},
         loop(Count);
      stop -> true
   end.
```

Registered Processes

- `register(atom(), pid()) -> true`
  - BIF

```
start() ->
   Pid = spawn(fun() -> loop(0) end),
   register(math_server, Pid),
   Pid.
```

```
math_server!{factorial, self(), 100}.
```

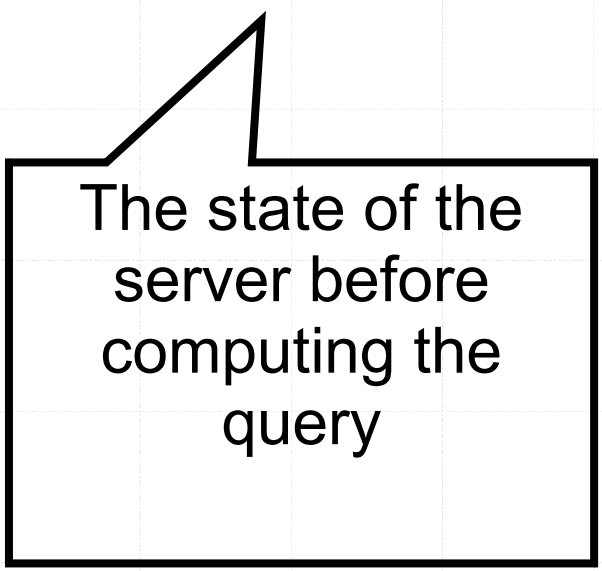
A Generic Server

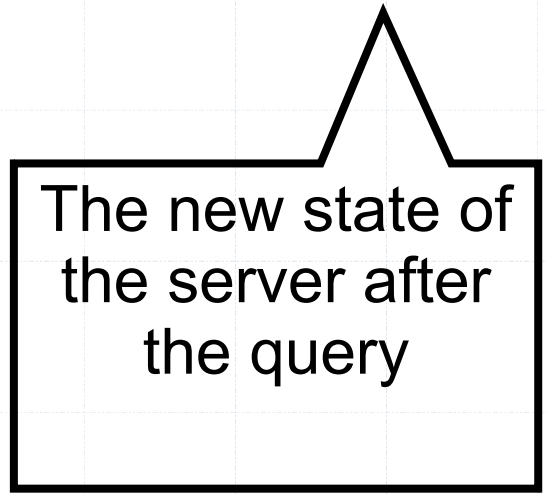
- Desired features
  - Proper reply behaviour - RPC
  - Parameterised by the “engine” F
  - Allows the engine to be “upgraded” dynamically
  - Robust: does not crash if the engine “goes wrong”

More Generic Client-Server

- Higher-order functions can be used to capture the design pattern of a client-server in a single module
  - The main engine of a server has “type”:

```
F:(State,Data) -> {Result,NewState}
```





Main Server Loop

```
loop(State, F) ->
    receive
        {request, From, Ref, Data} ->
            {R, NS} = F(State, Data),
            From ! {result, Ref, R},
            loop(NS, F);
        {update, From, Ref, NewF} ->
            From ! {ok, Ref},
            loop(State, NewF);
        stop -> true
    end.
```

Exceptions

- Expression evaluation can fail
- Examples:
  - Bad pattern matching (`{} = {1}`)
    - `**exited: {{badmatch,{1}},[{erl_eval,expr,3}]}**`
  - Arithmetic error (`1/0`).
    - `**exited: {badarith,…}**`
- Using `catch` expression
  - `catch(1/0) = {'EXIT',{badarith,…}}`
  - `catch(1/1) = 1`

More Robust Main Server Loop

```
loop(State, F) ->
  receive
    {request, From, Ref, Data} ->
       case catch(F(State, Data)) of
          {'EXIT', Reason} ->
             From!{exit, Ref, Reason},
             loop(State, F);
          {R, NewState} ->
             From!{result, Ref, R},
             loop(NewState, F)
       end;
    …
```

RPC – Client Interface

- Generic client with exception passing

```
request(Pid, Data) ->
    Ref = make_ref(),
    Pid!{request, self(), Ref, Data},
    receive
        {result, Ref, Result} ->
            Result;
        {exit, Ref, Reason} ->
            exit(Reason)
    end.
```

Generic Server Instance

```
-module(math_server).
-import(math_stuff,[factorial/1]).
-export([start/0]).

start() -> g_server:start(math_server,
                          0,
                          fun fx/2).
fx(Count, {factorial, N}) ->
    Result = factorial(N),
    {Result, Count+1};
fx(Count, get_count) ->
    {Count, Count}.
```

Generic Server Instance

- Encapsulated client interface

```
compute_factorial(Pid, N) ->
    g_server:request(Pid,
                     {factorial, N}).

get_count(Pid) ->
    g_server:request(Pid,
                     get_count).
```

Selective Receive

- Clauses can have guards
  - Guards must be composed from terminating functions (BIFs)

```
loop(Permits) ->
  receive
    {acquire,Pid,Ref} when Permits>0 ->
       Pid!{acquired, Ref},
       loop(Permits-1);
    release ->
       loop(Permits+1)
  end.
```

Distributed Execution

- Start Erlang node
  - `erl -sname node`
- Start another Erlang node
  - `erl -sname other`
- Send messages between nodes
  - It just works
  - For example to send to a registered process
    `{process, node@localhost}!message`

Distributed Example

- Start the `math_server` on one node
  - For example on `node@T60`
  - Run: `math_server:start().`
- Now, on another node we can off-load "heavy" factorial computation
  - For example on `other@T60`
  - Run:

```
math_server:compute_factorial(
        {math_server,node@T60},
        42).
```

Conclusions – Erlang

- Functional
- Concurrent
- Distributed
  - Basics