ERICSSON

CONCURRENT PROGRAMMING XXL

Industrial Use of Erlang – Introduction

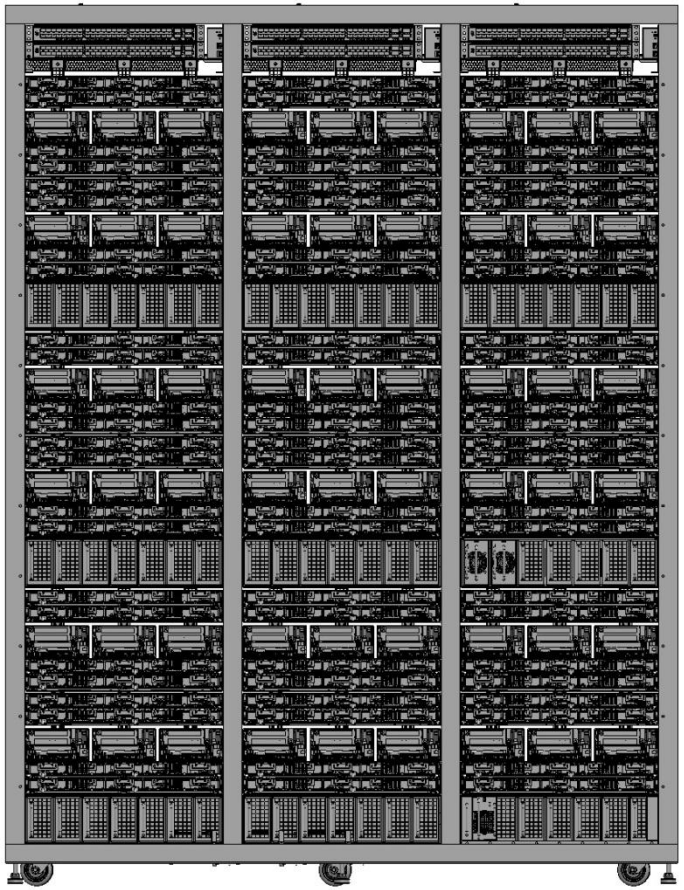
Karol Ostrovský (karol.ostrovsky@gmail.com)

MOTIVATION – MACHINE

- Ericsson Blade System
  - 3 sub-racks
  - 14 blades
    - 2 routers
    - 12 compute nodes
  - 6 core Intel x86
    - 12 SMT threads total
  - 432 simultaneously running processes
  - Backplane: 1 or 10Gbps

MOTIVATION – MACHINE

- Open Compute System
  - New OpenRack design
  - Triplet rack
  - 3 sub-racks
  - 18 dual-CPU blades
  - 8 core Intel x86
    - 16 SMT threads total
  - 2592 simultaneously running processes



MOTIVATION – THE TASK

› *Design on of the following systems:*

› Payment transaction handling
› Instant messaging back-end
› Multi-player game back-end
› Cloud infrastructure message-passing middleware
› Cloud routing/switching
› Telephone exchange

MOTIVATION – CONSIDER

- What are the main challenges?
- What distinguishes those systems from others (see the picture on the right)?
- What tools might be appropriate?

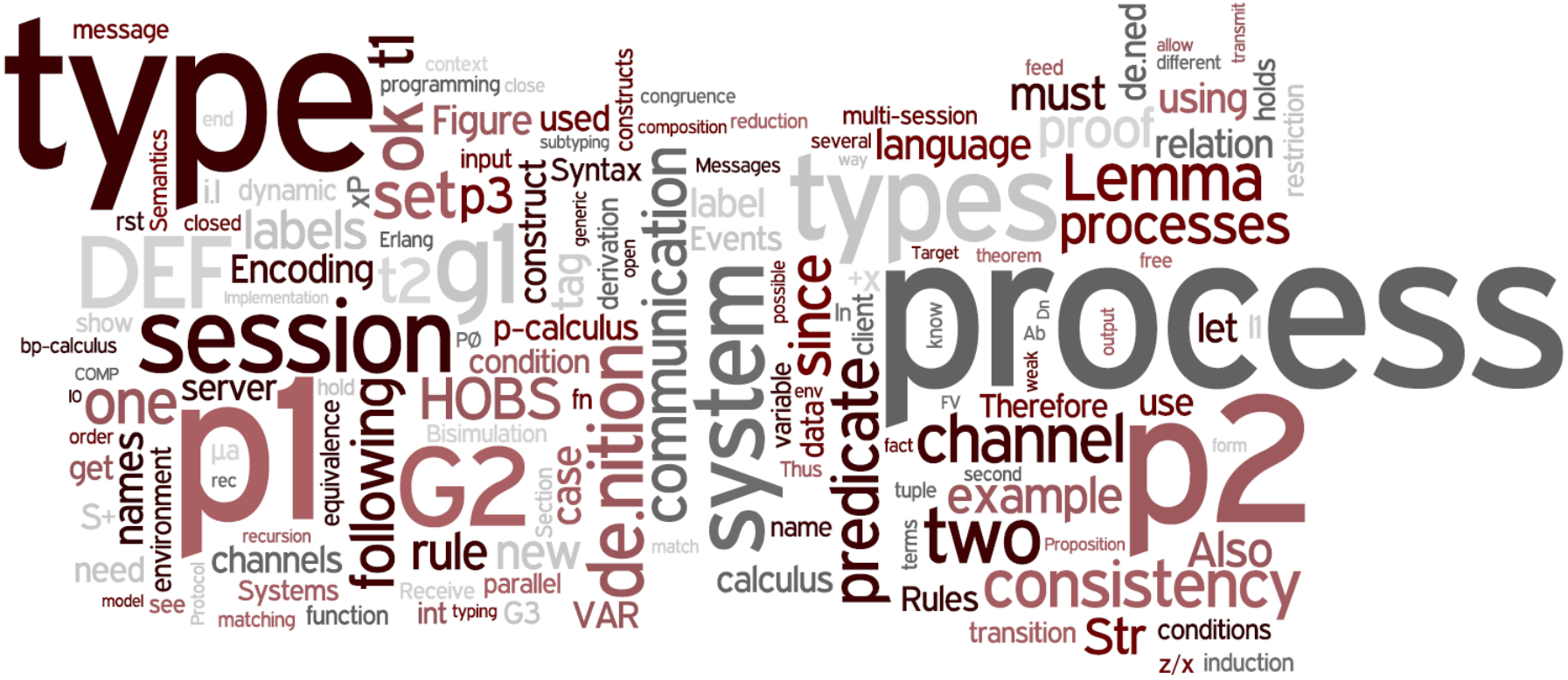
KAROL OSTROVSKÝ

- M.Sc. – Comenius University, Bratislava
- Ph.D. – Chalmers
- Post-doc – Chalmers
- System Designer – Dfind, on assignment at Ericsson
  - Operations & Maintenance Subsystem

MY CHALMERS YEARS

- Research in static analysis of concurrent programming languages
  - Type systems
  - Protocol analysis
- Main course responsible PPxT
  - Developed the course between 2005 and 2010

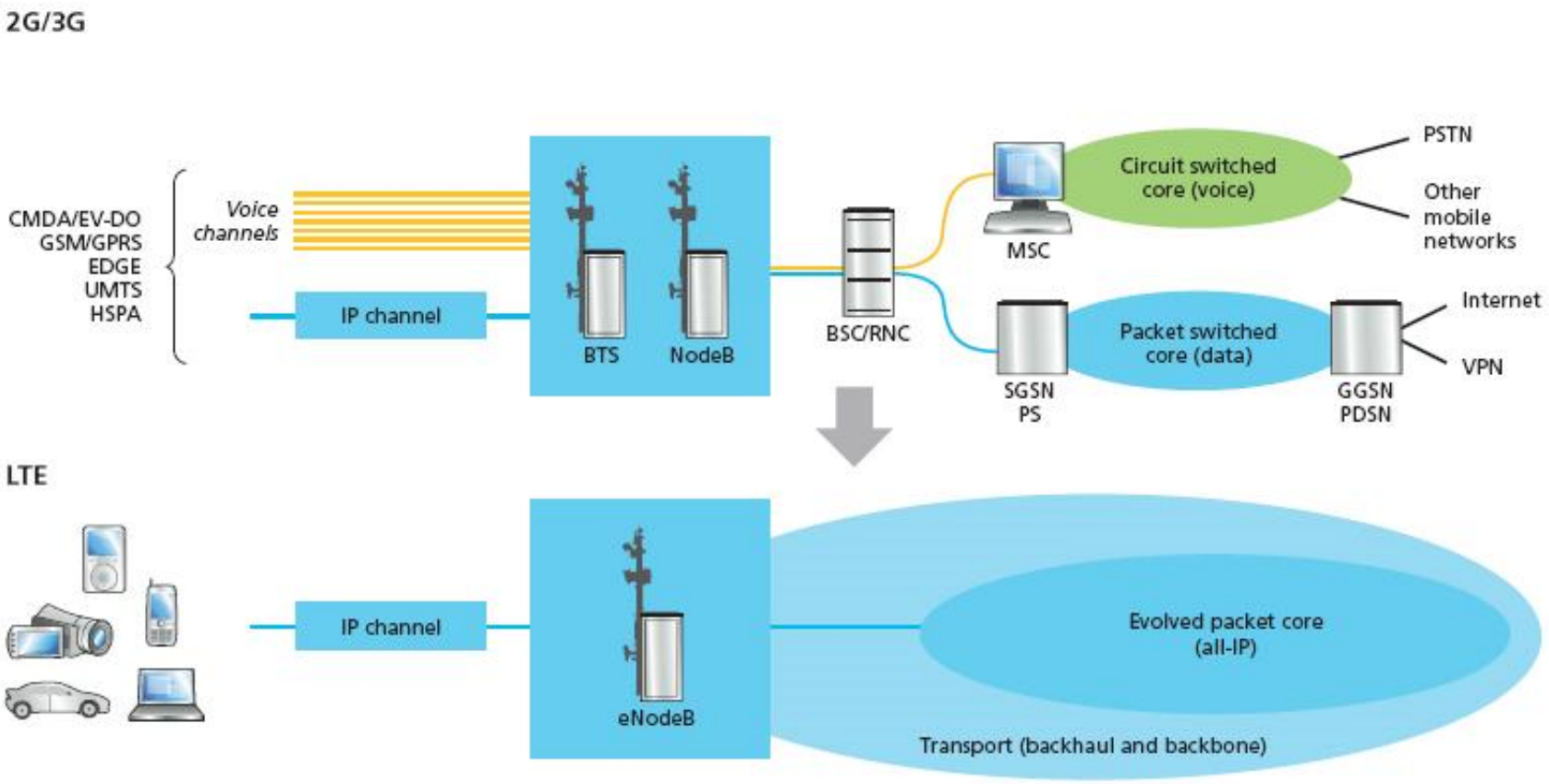
PHD THESIS


message
type
t1
context
programming
close
end
Semantics
ok
Figure
used
constructs
congruence
composition
reduction
multi-session
several
way
language
proof
must
feed
de.ned
allow
different
transmit
using
holds
restriction
relation
input
subtyping
Syntax
Messages
construct
set
p3
i.l
dynamic
xP
rst
closed
labels
Erlang
generic
derivation
open
label
Events
types
Lemma
processes
DEF
Encoding
t2
g1
tag
Implementation
Target
theorem
free
process
show
bp-calculus
session
PØ
p-calculus
condition
possible
since
In
client
+x
know
Dn
Ab
output
let
l1
COMP
server
hold
HOBS
fn
variable
env
data
predicate
FV
Therefore
use
one
p1
Bisimulation
channel
p2
order
get
names
environment
μa
rec
equivalence
following
G2
case
Section
Thus
fact
form
second
tuple
example
S+
need
channels
recursion
rule
new
de.nition
communication
match
name
two
Proposition
terms
Also
Protocol
model
see
Systems
matching
function
Receive
parallel
int
typing
G3
VAR
calculus
Rules
consistency
transition
Str
conditions
z/x
induction

KO@ERICSSON

- Business Unit Networks
  - Development Unit IP and Broadband
    - Product Development Unit Packet Core
      - SGSN-MME
        - *O&M sub-system*
        - 2G sub-system
        - 3G sub-system
        - ...
      - EPG
      - CPG
      - ...

MOBILE TELECOM NETWORK



PACKET CORE NETWORK

- 3GPP
  - Defines standards (mostly protocols)
  - Interoperability is essential
- SGSN-MME
  - Servicing GPRS Support Node (2G/3G)
  - Mobility Management Entity (4G)
  - Control signalling
    - Admission control, Authentication
    - Mobility, roaming
  - Payload transport (not in 4G)

CHALLENGES

- Open *soft* real-time system
- Resilience
  - HW failures
  - SW failures
    - External
    - Internal
- Appropriate tools?

ENTER ERLANG

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

RECURSION

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

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

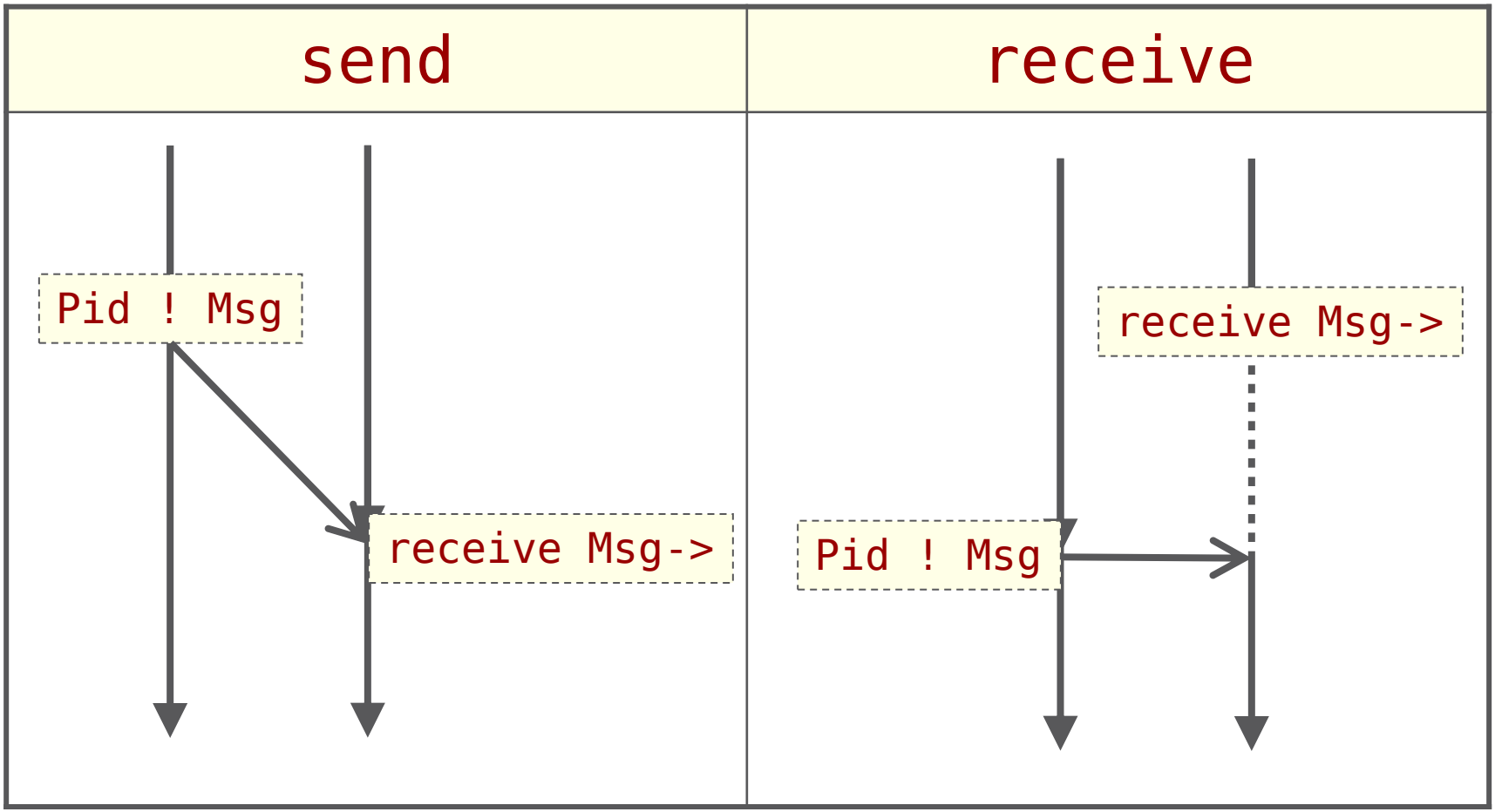
CONCURRENCY

- Based on Message Passing:
  - Q: What form of synchronisation?
  - A: Asynchronous

  - Q: What form of process naming?
  - A: Direct, asymmetric

ASYNCHRONOUS MP

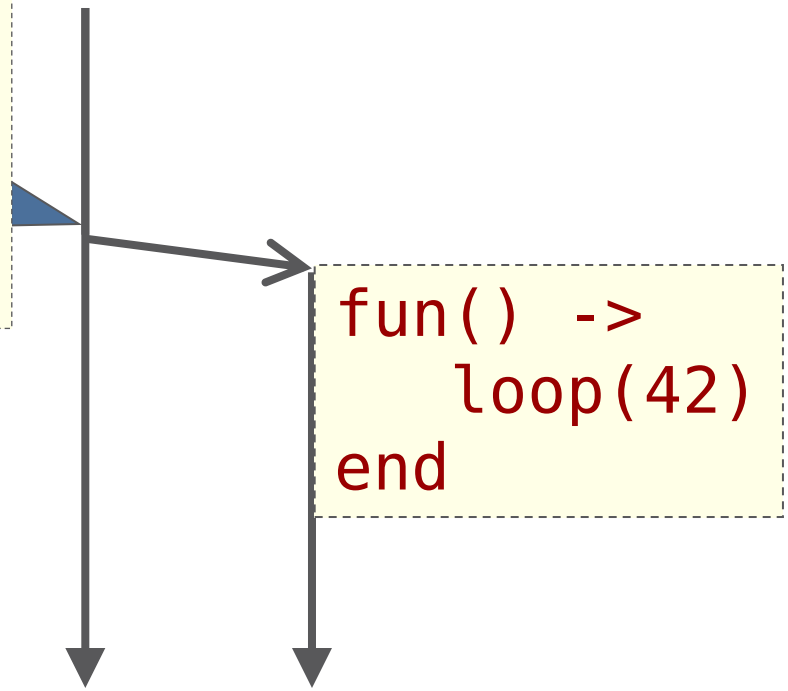
› Asynchronous send, receive from mailbox



CONCURRENT EXECUTION

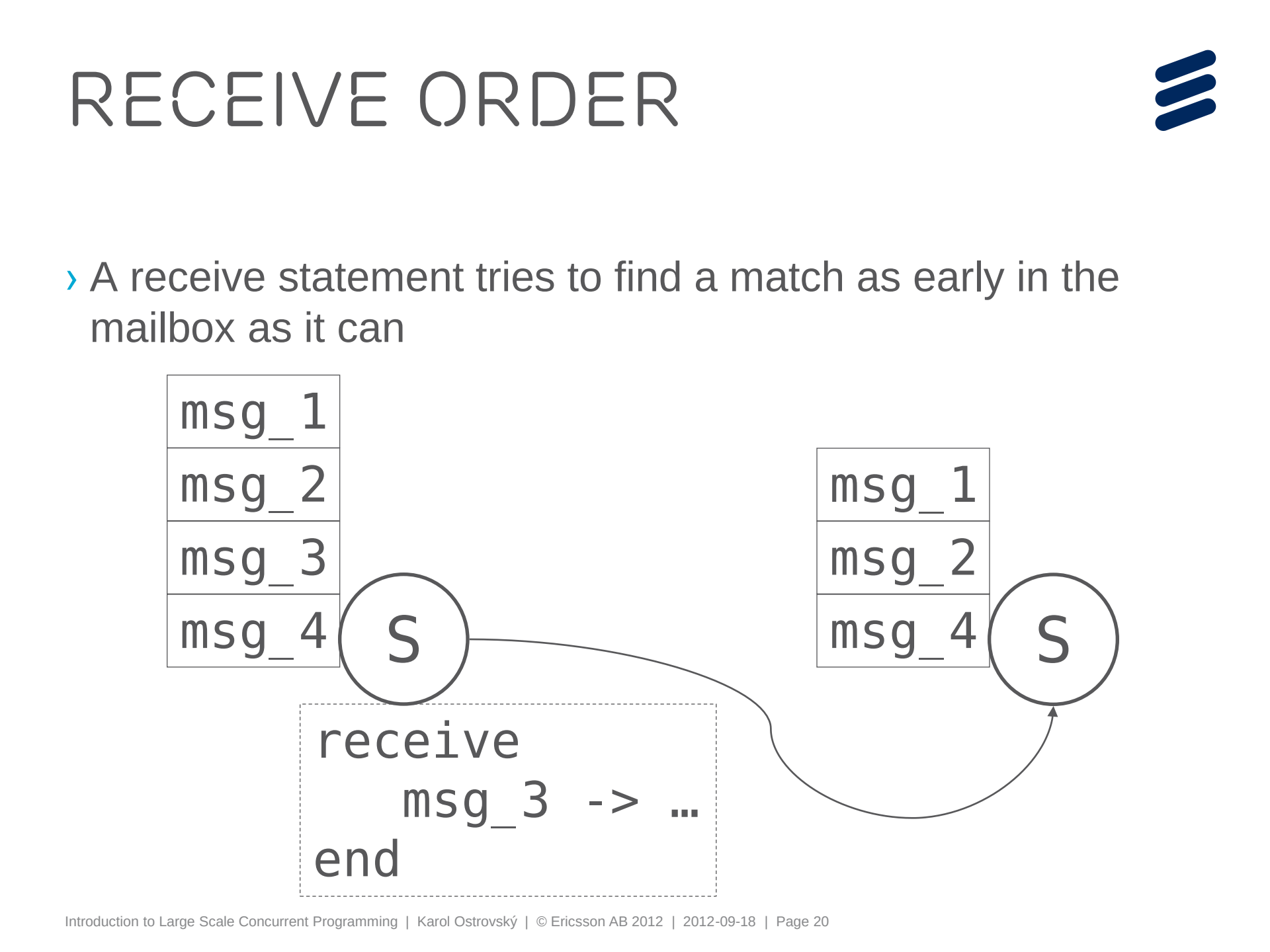
› Spawn a function evaluation in a separate thread

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



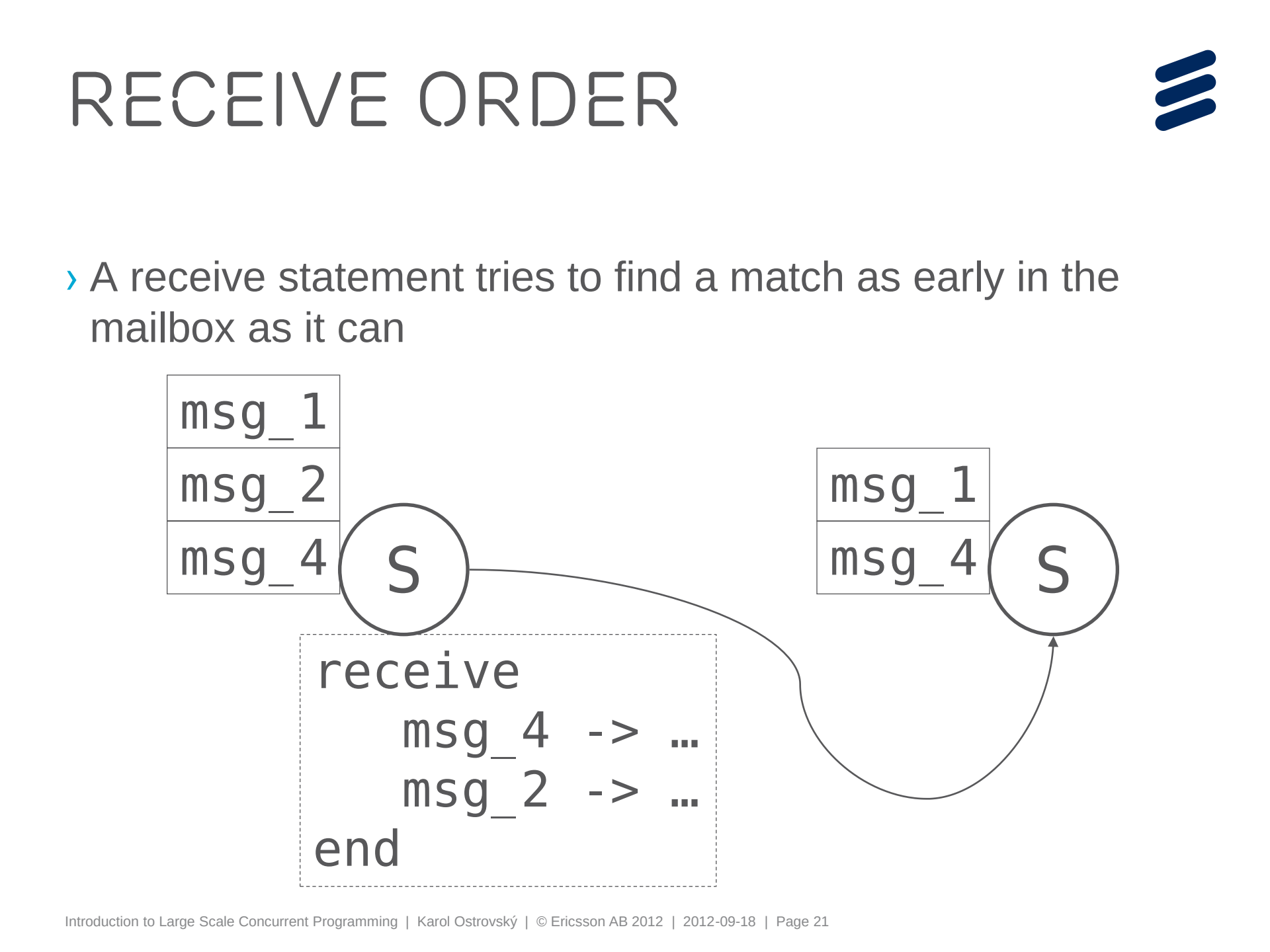
› Function must use constant space
  – Tail recursive

RECEIVE ORDER



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

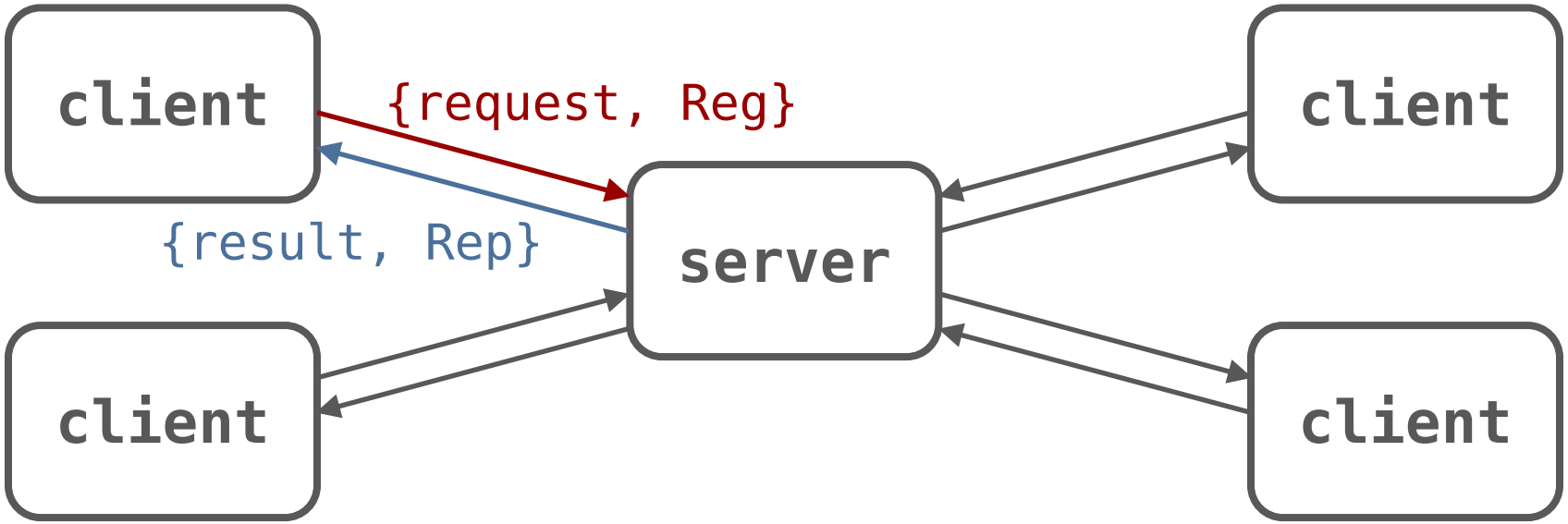
RECEIVE ORDER



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

CLIENT-SERVER

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



MODELLING C-S

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

```
-module(math_server).
-export([start/0]).
-export([compute_factorial/2,
         get_count/2]).
-export([loop/1]).

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

%%continues
```

MAIN SERVER LOOP

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

A GENERIC SERVER

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

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}
```

The state of the server before computing the query

The new state of the server after the query

GENERIC 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.
```

CLIENT INTERFACE

› Generic client can make a generic request

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

REGISTERED PROCESSES

- Centrally register a process under a name
  - BIF: `register(atom(), pid()) -> true`

```
start(Name, State, F) ->
    Pid = spawn(fun() ->
                    loop(State, F)
                end),
    register(Name, Pid),
    Pid.
```

REGISTERED PROCESSES

› Sending to a registered process

```
math_server ! stop.
```

› For registered processes the name is the same as its pid
  – Almost (subtle differences in error handling)

SERVER INSTANCE

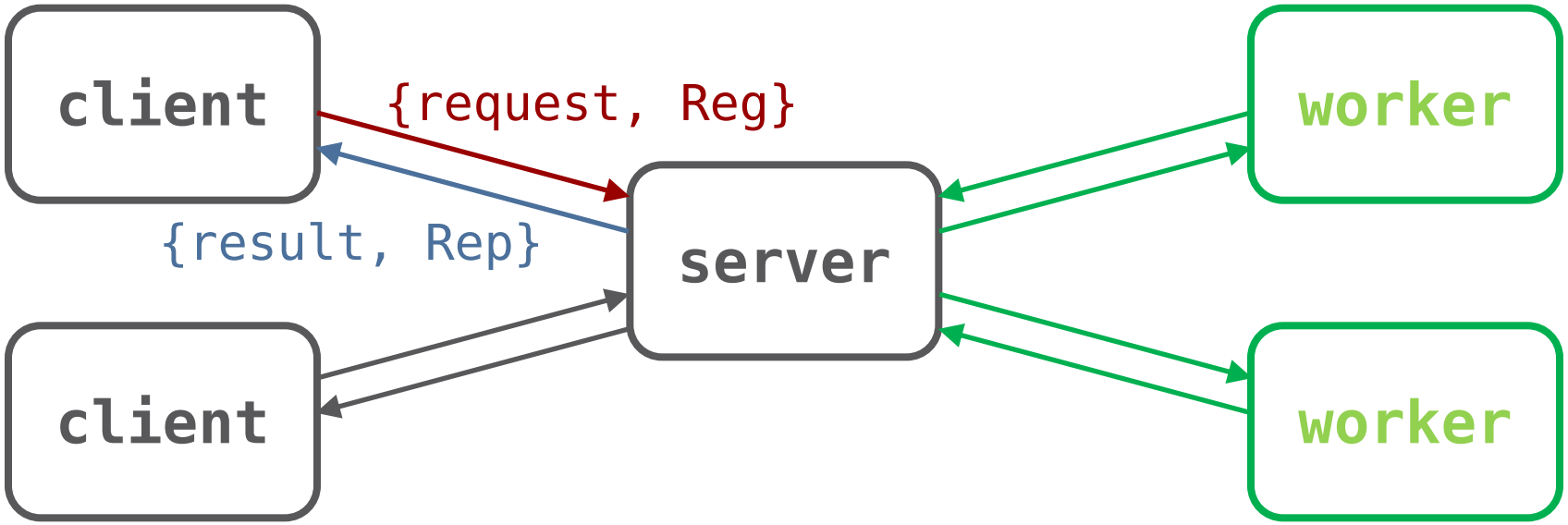
```
-module(g_math_server).
-export([start/0]).
-export([compute_factorial/1, get_count/0]).

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}.
%%continues
```

SERVER INSTANCE

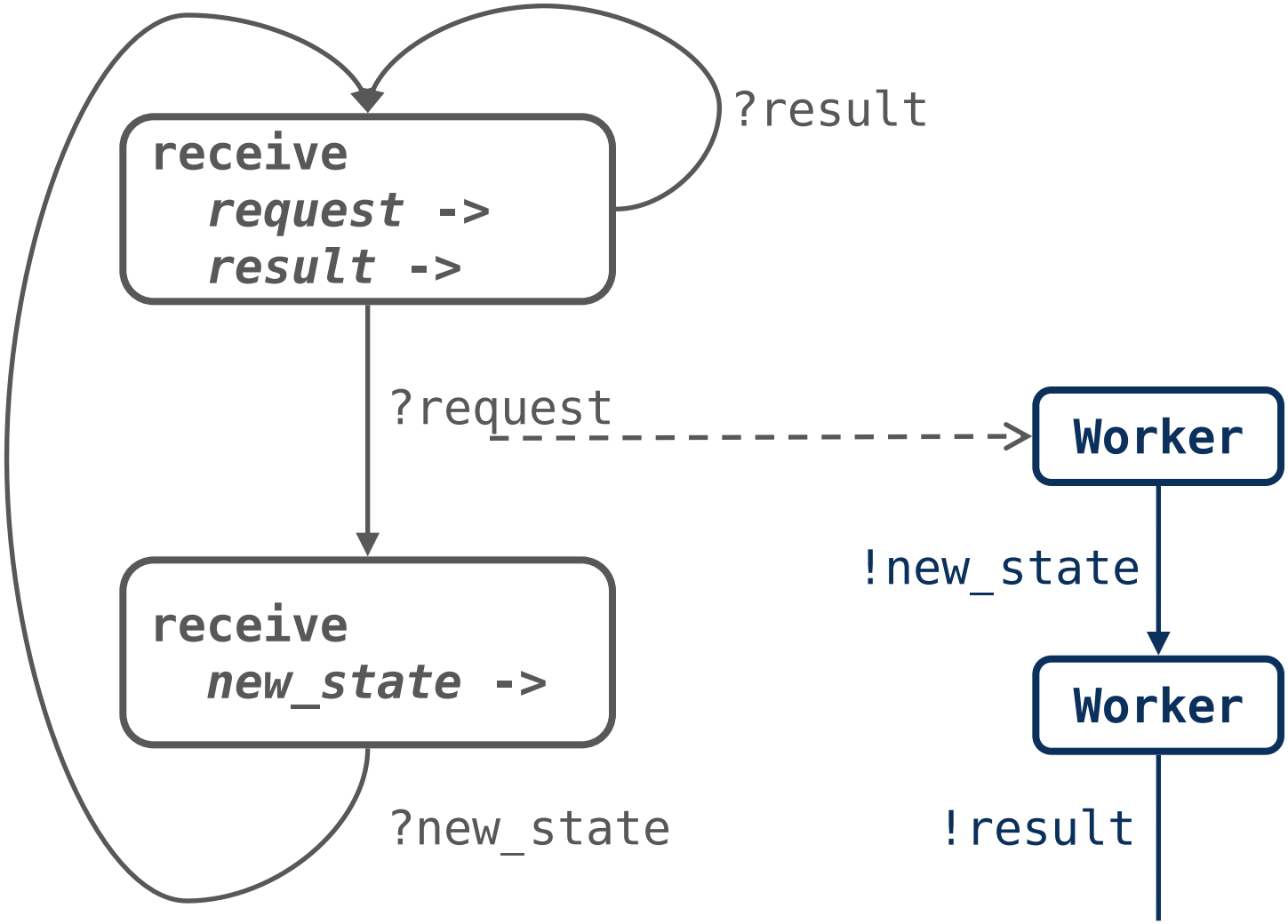
```
%%continuation

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


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

PARALLELISED SERVER

- Compute the heavy computations in separate worker processes
  - New engine function
  - New error handling issues



STATE DIAGRAM



NEW ENGINE FUNCTION

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

```
F(State, Data, SPid, Ref) ->
 Proc()⟨
   SPid!{new_state,Ref,NewState},
   SPid!{result,Ref,Result}⟩
```

PARALLEL SERVER LOOP

```
loop(Count, F, Pending) ->
  receive
    {request, From, Ref, Data} ->
      Self = self(),
      Pid = spawn(
          fun() ->
            F(State, Data, Self, Ref)
          end),
      receive
        {new_state, Ref, NewState} ->
          ?MODULE:loop(NewState, F,
           [{From,Ref,Pid}|Pending]);
      end;
```

PARALLEL SERVER LOOP

```
  {result, Ref, Result} ->
     case keysearch(Ref, 2, Pending) of
       {value, {From, Ref, _}} ->
          From!{result, Ref, Result},
          ?MODULE:loop(State, F,
                       keydelete(Ref,2,Pending));
       false ->
          ?MODULE:loop(State,F,Pending)
     end;
  %% update and stop as before
end.
```

Module lists

DYNAMIC CODE UPDATE

- We want to update the running math server from V1 to the parallelised V2
  - Compile and load new version,
  - Send any message,
  - And the new loop code will be called
  - Problems?
    - New engine function type
    - New internal state (pending list)
    - loop/2 versus loop/3

DYNAMIC CODE UPDATE

› Updating the loop between versions

```
loop(OldState, OldF) ->
  io:format("This is V2 starting...~n",[]),
  loop(OldState,
       fun(State, Data, SPid, Ref) ->
         {Result,NewState} = OldF(State,Data),
         SPid!{new_state, Ref, NewState},
         SPid!{result, Ref, Result}
       end,
       []).
```

NEW MATH ENGINE

```
fx(Count, {factorial, N}, SPid, Ref) ->
    SPid!{new_state, Ref, Count+1},
    SPid!{result, Ref, factorial(N)};
fx(Count, get_count, SPid, Ref) ->
    SPid!{new_state, Ref, Count},
    SPid!{result, Ref, Count}.

update_fx() ->
    g_server:update(math_server,
                    fun fx/4).
```

ERLANG/OTP

- Framework for defining applications
- Defines several general behaviours, examples:
  - Servers
  - Finite state machines
  - Event handlers
  - Supervisors
  - Applications
- Behaviour captured once and for all
  - Remains to define application-specific functions,
  - Which are often without concurrency problems

ERLANG/OTP

- The main source
  - http://www.erlang.org/
- Excellent learning resource
  - http://learnyousomeerlang.com/
- Plenty of useful blogs and mailing lists

ERLANG – STRENGTHS

- Well suited for
  - Control handling of telecom traffic
  - Application layer (OSI model) applications
    - Web servers, etc.
  - Domain Specific Language – framework
    - Test scripting
- Reasonably high-level (as compared to for example C)
  - Good for software maintenance
- OTP – includes useful building blocks
  - Supervisor
  - Generic server
  - Finite state machine

ERLANG – WEAKNESSES

- A bit too low-level language
  - Given current HW limitations one must sometimes optimise to the point where the code is not portable (with the same performance)
  - Raise the abstraction and provide a customisable compiler, VM
- Where is the type system?
  - A static type system of Haskell-nature would be a hindrance
  - But good static analysis tools are desperately needed

THE MISSING SUBJECT

- Software maintenance
  - Software lives long
    - Especially telecom systems (decades)
    - Banking systems live even longer (think COBOL)
  - People change
  - Organisations change
  - Hardware changes
  - Requirements change
  - Documentation often *does not* change

MAINTENANCE

- The developer's challenge
  - Write simple (readable) and efficient code:
    1. Write a straightforward and working solution first
    2. Optimise later (or even better skip this step)
- Think smart but do not over-optimise
  - Optimisations complicate maintenance
- The code is often the only reliable document
  - Types can be very good documentation

SYNTHESIS VS. ANALYSIS

- Emphasis on synthesis so far
  - Software development
- Around 30% of manpower is spent on testing
  - Analytical work
  - Do you like to break a system?
- But testing can also be “synthetical”
  - Testing frameworks
    - Quickcheck
    - SGSN-MME has its own
  - Would you like to formally prove the system correct?

MORE THAN TRUE

Sayings

- The greatest performance improvement of all is when a system goes from not-working to working
- The only thing worse than a problem that happens all the time is a problem that doesn't happen all the time

MY FAVOURITE

The Message from this Course

- Should you forget everything from this course, please, remember at least this saying:

> Use the right tool for the job.

CHALMERS GÖTEBORG UNIVERSITY 3

ERICSSON