



ERLANG

Functional Programming in industry

Leslaw Lopacki
leslaw@lopacki.net

Courtesy of Urban Boquist and Christer Nilsson (Ericsson Gothenburg)

Outline

- ▶ Mobile Telecommunications Networks
- ▶ Packet Core Network – GPRS, UMTS & SGSN
- ▶ Use of Erlang in SGSN
- ▶ SGSN Design Principles for Erlang:
 - concurrency
 - distribution
 - fault tolerance
 - overload protection
 - runtime code replacement
- ▶ Erlang basics and examples

Mobile Telecommunications Networks - GSM

Services in telecommunications networks:

CS – circuit switched

- voice
- SMS

PS – packet switched

- everything that is “IP”
- wap / www
- email
- MMS

GPRS - General Packet Radio Service

Packet Core Network

Radio Network

Packet Core Network

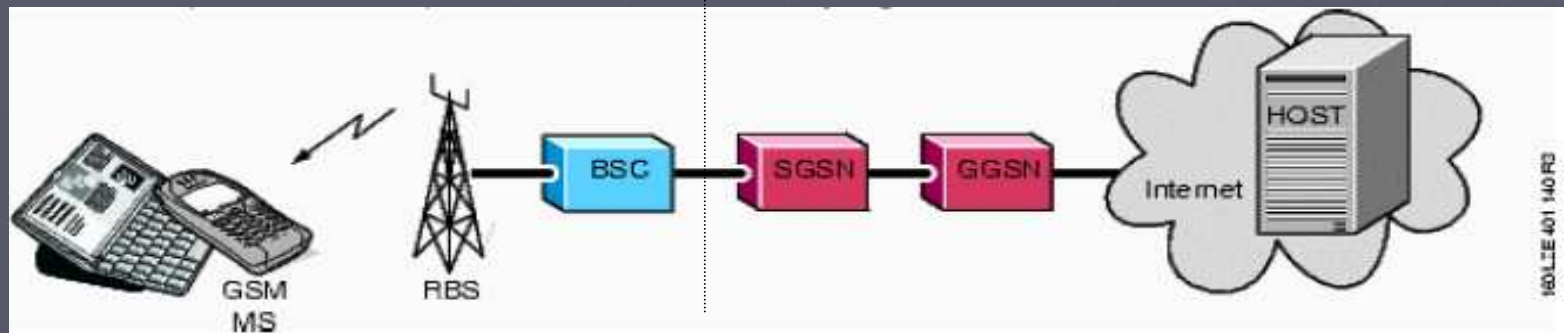


Figure: User Plane through the GSM network

- ▶ GSN (GPRS Support Network) nodes:
 - SGSN – Serving GSN
 - GGSN – Gateway GSN
- ▶ Basic throughput:
 - Up to 115 kbps with GPRS
 - Up to 240 kbps with EDGE – Enhanced Data Rates for GSM Evolution

PCN in "3G" and "Turbo-3G" – WCDMA and HSDPA

- ▶ Different Radio Network
- ▶ Packet Core Network (almost) the same as the one in GPRS
- ▶ Ericsson SGSN is "dual access" – GPRS and WCDMA in one
- ▶ Much higher (end user) speeds:
 - Up to 384 kbps for 3G (WCDMA)
 - Up to 14.4 Mbps for HSDPA (later up to 42 Mbit – Evolved HSPA)
- ▶ Voice / video calls are still CS!
- ▶ Streaming video is PS
(TV == MBMS – Multimedia Broadcast Multicast Service)
- ▶ Future: voice / video in PS
- ▶ "Voice-over-IP"

Ericsson SGSN Node

Capacity

- ~ 50 k subscribers, 2000
- ~ 100 k subscribers, 2002
- ~ 500 k subscribers, 2004
- ~ 1 M subscribers, 2005
- ~ 2 M subscribers, 2008



SGSN – Basic Services

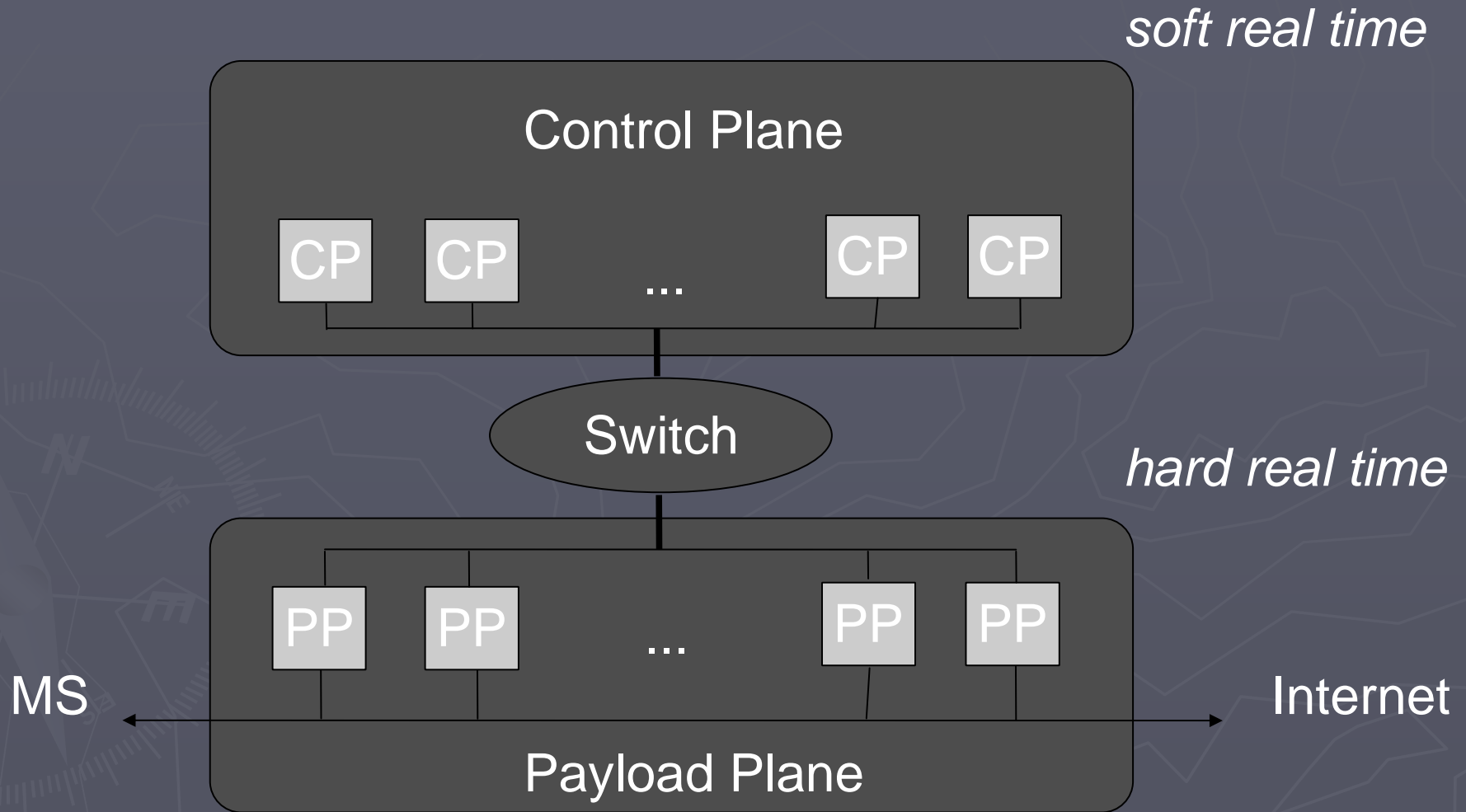
Control Signalling

- ▶ authentication
- ▶ admission control
- ▶ quality of service
- ▶ mobility
- ▶ roaming
- ▶ ...

Payload transport

- ▶ user traffic
- ▶ charging

SGSN Architecture



SGSN Hardware

- ▶ ≈ 20-30 Control Processors (boards):
 - UltraSPARC or PowerPC CPUs
 - 2 GB memory
 - Solaris/Linux + Erlang / C / C++
- ▶ ≈ 20-30 Payload Processors (boards):
 - PowerPC CPUs
 - Special hardware (FPGAs) for encryption
 - Physical devices: frame relay, atm, ...
 - VxWorks + C / C++
- ▶ Backplane: 1 Gbit Ethernet

SGSN Control Signalling

- ▶ attach (phone is turned on)
- ▶ israu (routing area update, mobility in radio network)
- ▶ activation (initiate payload traffic)
- ▶ etc. [hundreds of signals]

Telecom standards are HUGE (see www.3gpp.org)!

We need a high level language – concentrate on GPRS, not on programming details!

Erlang/OTP

- ▶ Invented at Ericsson Computer Science Lab in the 1980s.
- ▶ Intended for large scale reliable telecom systems.
- ▶ Erlang is:
 - functional language
 - with built-in support for concurrency
- ▶ OTP (Open Telecom Platform)
== Erlang + lots of libraries.

Why Erlang?

- ▶ Good things in Erlang:
 - built-in concurrency (processes and message passing)
 - built-in distribution
 - built-in fault-tolerance
 - support for runtime code replacement
 - a dynamic language
 - a dynamically typed language
- ▶ This is exactly what is needed to build a robust Control Plane in a telecom system!

In SGSN:

- ▶ Control Plane Software is not time critical (Erlang)
- ▶ User Plane (payload) is time critical (C)

Erlang – Concurrency

- ▶ “Normal” synchronization primitives - semaphores or monitors
 - does not look the same in Erlang
 - instead everything is done with processes and message passing.
- ▶ Mutual exclusion:
 - use a single process to handle resource
 - clients call process to get access.
- ▶ Critical sections:
 - allow only one process to execute section

Erlang - Distribution

- ▶ General rule in SGSN:
 - avoid remote communication or synchronization if possible
- ▶ Design algorithms that work independently on each node:
 - fault tolerance
 - load balancing
- ▶ Avoid relying on global resources
- ▶ Data handling:
 - keep as much locally as possible (typically traffic data associated with mobile phones)
 - some data must be distributed / shared (e.g. using mnesia)
 - many different variants of persistency, redundancy, replication

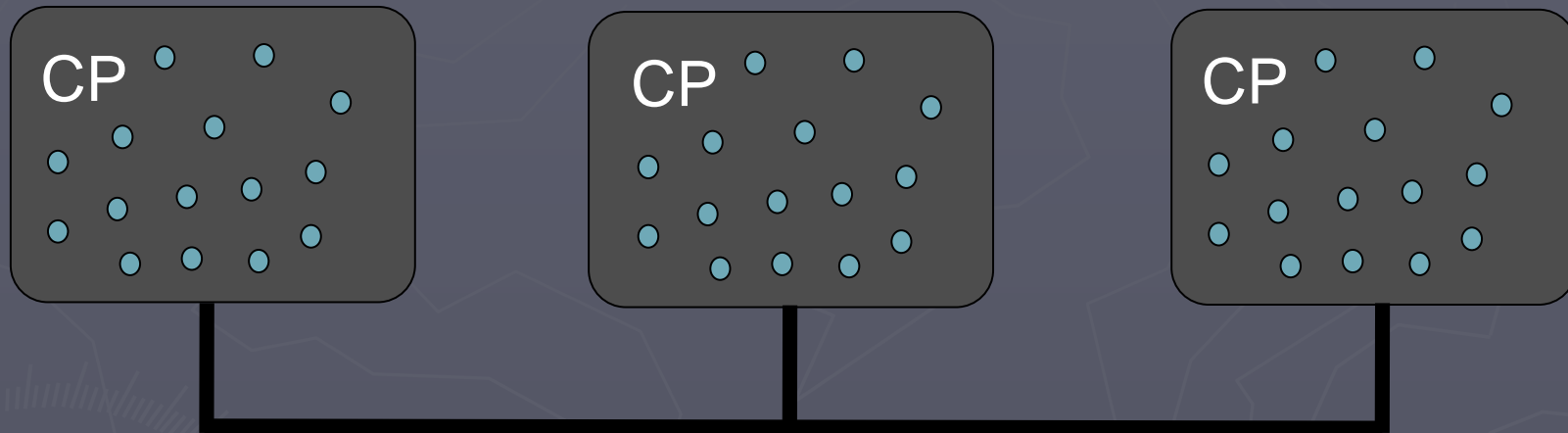
Fault Tolerance

- ▶ SGSN must never be out-of-service! (99.999%)
- ▶ Hardware fault tolerance
 - Faulty boards are automatically taken out of service
 - Mobile phones automatically redistributed
- ▶ Software fault tolerance
 - SW error triggered by one phone should not affect others!
 - Serious error in "system SW" should affect at most the phones handled by that board (not the whole node)

How can such requirements be realized?

Example: the SW handling one phone goes crazy and overwrites all the memory with garbage.

SGSN Architecture – Control Plane



- ▶ On each CP \approx 100 processes providing “system services”
 - “static workers”
- ▶ On each CP \approx 50.000 processes each handling one phone
 - “dynamic workers”

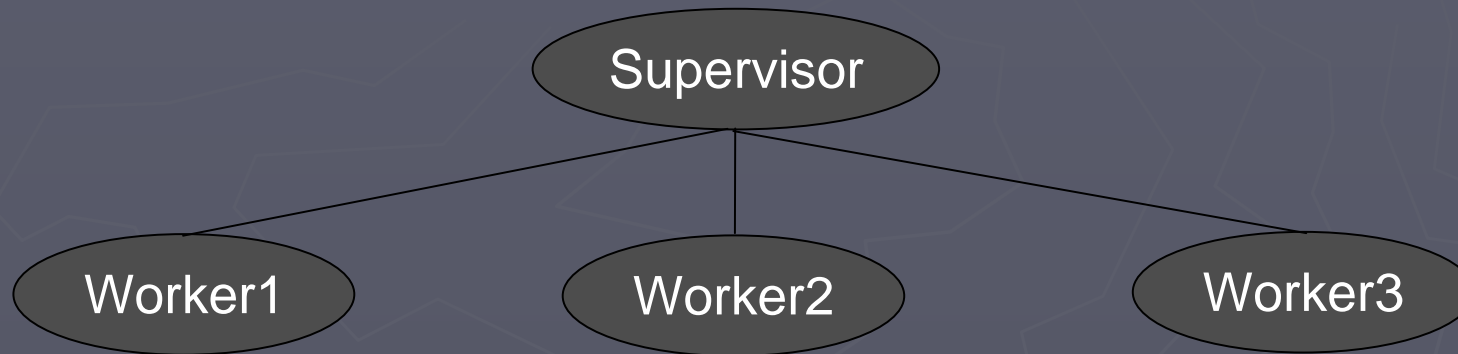
Dynamic workers

- ▶ System principle:
 - one Erlang process handles all signalling with a single mobile phone
- ▶ When a signal received in payload plane:
 - payload plane translates a “signal” from the mobile phone into an Erlang message
 - then sends it to the correct dynamic worker, and vice versa
- ▶ A worker has a state machine:
 - receive a signal – do some computation – send a reply signal
 - a little bit like an Entity Bean in J2EE

Dynamic workers cont.

- ▶ A process crash should never affect other mobiles:
 - Erlang guarantees memory protection
- ▶ SW errors in SGSN:
 - lead to a short service outage for the phone
 - dynamic worker will be restarted after the crash
- ▶ Same for SW errors in MS:
 - e.g. failure to follow standards will crash dynamic worker (offensive programming)

Supervision and Escalation



- ▶ Crash of worker is noticed by supervisor
- ▶ Supervisor triggers "recovery action"
- ▶ Either the crashed worker is restarted
or
- ▶ All workers are killed and restarted

Runtime code replacement

- ▶ Fact: SW is never bug free!
- ▶ Must be able to install error corrections into already delivered systems without disturbing operation
- ▶ Erlang can load a new version of a module in a running system
- ▶ Be careful!
Code loading requires co-operation from the running SW and great care from the SW designer

Overload Protection

- ▶ If CPU load or memory usage goes to high SGSN will not accept new connections from mobile phones
- ▶ The SGSN must never stop to “respond” because of overload, better to skip service for some phones
- ▶ Realized in message passing - if OLP hits messages are discarded:
 - silently dropped
 - or a denial reply generated

Erlang basic syntax

- ▶ Erlang shell :

```
erl
```

- ▶ Modules and Functions:

```
-module(my_mod).  
-export(double/1).
```

```
double(X) -> 2 * X.
```

- ▶ Calling double/1:

```
my_mod:double(4).
```

- ▶ Atoms:

```
cat, dog, home, a2 ..
```

- ▶ Tuples :

```
{1,2,cat,home}
```

- ▶ Lists :

```
[{1,2,cat,home},1,2,3]
```

- ▶ Variables :

```
A = {2,3,horse,stable}.
```

```
B = [{1,2,cat,home},1,2,3].
```

```
Var = [A|B].
```

- ▶ Writing to output:

```
io:format("Hello world").
```

Erlang syntax - case and functional clause

- ▶ Case clause - case and pattern matching:

```
...  
Loc =  
  case Var of  
    {_,_,cat,X} -> io:format("Hello Cat"),X;  
    {_,_,horse,X} -> io:format("Hello Horse"),X;  
    _ -> io:format("No entrance"),none  
  end.  
...
```

- ▶ Function clause:

```
...  
hello({_,_,cat,X}) -> io:format("Hello Cat"),X;  
hello({_,_,horse,X}) -> io:format("Hello Horse"),X.  
hello(_) -> io:format("No entrance"),none.  
...
```

Erlang syntax - Recursion

► Simple:

```
-module(fact).  
-export([fact1/1]).
```

```
fact1(0) ->  
    1;  
fact1(N) ->  
    N*fact1(N-1).
```

► Optimal - tail recursive:

```
-module(fact).  
-export([fact2/1]).
```

```
fact2(N) ->  
    fact2(N,1).  
fact2(0,A) ->  
    A;  
fact2(N,A) ->  
    fact2(N-1,N*A).
```

Erlang advanced syntax

► Dynamic code:

```
...
Fun = fun(Var)
  case Var of
    {_,_,cat,X} -> io:format("Hello Cat"),X;
    {_,_,horse,X} -> io:format("Hello Horse"),X;
    _ -> io:format("Not welcome here"),none
  end.
...
```

Calling Fun:

```
Fun({1,2,cat,home}).
```

Passing Fun to another function:

```
call_fun(Fun,[]) -> ok;
call_fun(Fun,[X|T]) -> Fun(X), call_fun(Fun,T).
...
List = [{1,2,cat,home},{2,3,horse,stable}].
call_fun(Fun,List).
```

Erlang message passing

sender:

...

Pid ! Msg,

...

receiver:

...

receive

Msg ->

<action>

end,

...

Example cont. - gen_server

sender:

...

```
Ret = gen_server:call(Pid, Msg),
```

...

receiver:

```
handle_call(Msg) ->
```

```
  case Msg of
```

```
    {add, N} ->
```

```
      {reply, N + 1};
```

```
    ...
```

```
  end.
```

What about “functional programming”?

- ▶ Designers implementing the GPRS standards should not need to bother with programming details.
- ▶ Framework code offers lots of “abstractions” to help out.
- ▶ Almost like a DSL (domain specific language).
- ▶ To realize this, functional programming is very good!
- ▶ But to summarize: FP is a great help – but not vital. Or?

Conclusions

Pros:

- ▶ Erlang works well for GPRS traffic control handling
- ▶ High level language – concentrate on important parts
- ▶ Has the right capabilities:
 - fault tolerance
 - distribution
 - ...

Cons:

- ▶ Hard to find good Erlang programmers
- ▶ Erlang/OTP not yet a main stream language
 - Insufficient programming environments (debugging, modelling, etc)
 - Single implementation maintained by too few people - bugs
- ▶ High level language – easy to create a real mess in just a few lines of code...

Links and References

- ▶ Erlang site:

<http://www.erlang.org>

- ▶ Erlang User Conference (Nov 2008)

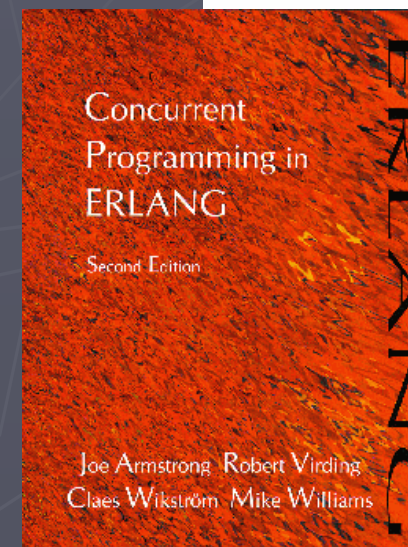
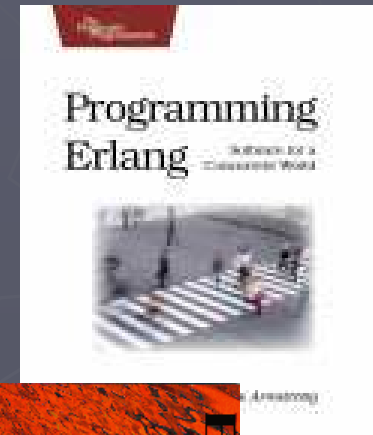
- ▶ Erlang Community:

<http://trapexit.org>

- ▶ Erlang group on LinkedIn

Books

- ▶ J. Armstrong
“*Programming Erlang*”
- ▶ J. Armstrong, R. Virding, C. Wikström,
M. Williams
“*Concurrent Programming in Erlang*”



Questions?

