FP in industry - Erlang

Urban Boquist – Ericsson AB
<Urban.Boquist@ericsson.com>

Outline

• Who Am I
• Mobile Telecommunications Networks
• Packet Core Network – GPRS & SGSN
• Use of Erlang in SGSN
• SGSN Design Principles for Erlang:
  – concurrency & distribution
  – fault tolerance
  – multicore
  – overload protection
  – runtime code replacement
• Examples

Who Am I?

• Chalmers (D-linjen)
• Chalmers (PhD, Compilation & Optimization of Haskell)
• Carlstedt Research & Technology (consultant)
• QEP (own startup, consultant)
• Ericsson AB, Lindholmen
• ...

GSM – GPRS

Services in telecommunications networks:

<table>
<thead>
<tr>
<th>CS – circuit switched</th>
<th>PS – packet switched</th>
</tr>
</thead>
<tbody>
<tr>
<td>voice</td>
<td>everything that is “IP”</td>
</tr>
<tr>
<td>SMS</td>
<td>www</td>
</tr>
<tr>
<td></td>
<td>email</td>
</tr>
<tr>
<td></td>
<td>MMS</td>
</tr>
</tbody>
</table>

• GPRS – General Packet Radio Service

GPRS

Radio Network  Packet Core Network

Figure 1: User Plane through the GSM network

• GSN (GPRS Support Network) nodes:
  • SGSN – Serving GSN
  • GGSN – Gateway GSN
• Basic throughout:
  • Up to 115 Kbps with GPRS
  • Up to 240 Kbps with EDGE – Enhanced Data Rates for GSM Evolution

“3G” – UMTS / WCDMA

• Different Radio Network
• Packet Core Network (almost) the same as in GPRS
• Ericsson SGSN is “dual access”
• 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 audio / video is PS (TV == MBMS)
• Future: voice / video in PS
• “Voice-over-IP”
3GPP

- Standards define everything.
- Interoperability is vital!
- "Tens of thousands" pages of standard text needed to build an SGSN.
- See www.3gpp.org.

SGSN – Basic Services

- Control Signalling
  - authentication
  - admission control
  - quality of service
  - mobility
  - roaming
  - ...

- Payload transport
  - user traffic
  - charging

SGSN Node

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

SGSN Architecture

- Control Plane
  - CP
  - CP
  - CP
  - CP

- Payload Plane
  - MS
  - PP
  - PP
  - PP

- Switch

- Internet

SGSN Hardware

- ~ 20-30 Control Processors (boards):
  - UltraSPARC or PowerPC cpus
  - 2 GB memory
  - Solaris/Linux + Erlang / C / C++
- ~ 20-30 Payload Processors (boards):
  - 1-3 PowerPC cpus
  - Special hardware (FPGAs) for encryption
  - Physical devices: frame relay, atm, ...
  - VxWorks + C / C++
- Backplane: 1 Gbit ethernet

Current release: ~ 2.000.000 Simultaneously Attached Users (phones)
Traffic Control in SGSN

- Control Processors (Solaris / Sparc or Linux / PowerPC)
- Most control signalling handled by Erlang code
- One “Erlang” running on each CP
- Distributed Erlang system with 20-40 nodes
- Mobile Phones are distributed over CP:s

Control Signalling

- attach (phone is turned on)
- israu (routing area update, mobility in radio network)
- activation (initiate payload traffic)
- etc. [thousands 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 + built-in support for concurrency.
- OTP (Open Telecom Platform) = Erlang + lots of libraries.

Erlang vs. Haskell

- Erlang can do most things Haskell can (pattern matching, higher order functions, list comprehensions, …)
- BUT – where Haskell is “beautiful”, Erlang is “ugly”!
- Erlang is strict (like ML, expressions evaluated immediately, not when they are needed)
- Erlang has no real type system (like LISP, everything compiles but may crash at runtime)

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
- This is exactly what is needed to build a robust Control Plane in a telecom system!
- Control Plane Software is not time critical (Erlang)
- User Plane (payload) is time critical (VxWorks + C)

Fault Tolerance

- SGSN must never be out-of-service! (99.999%)
- Hardware fault tolerance
  - Faulty boards are automatically taken out of service
  - Mobile phones 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

Think: 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 ≈ 100 processes providing "system services"
  – "static workers"
• On each CP ≈ 50,000 processes each handling one phone
  – "dynamic workers"

Dynamic workers

• System principle: one Erlang process handles all signalling with a single mobile phone
• A worker encodes a number of state machines: receive a signal – do some computation – send a reply signal
• Payload plane translates a "signal" from the mobile phone into an Erlang message and sends it to the correct dynamic worker, and vice versa

Dynamic workers cont.

• A process crash should never affect other mobiles (Erlang guarantees memory protection)
• SW errors in SGSN leads 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

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

Recovery principles

• Recovery action after SW crash is "restart"
• Many restart levels:
  – very very small restart
  – very small restart
  – small restart
  – medium restart
  – large restart
  – SGSN restart
• Lowest restart level affects only one mobile phone
• Highest level affects all phones
• Try low level first, if it does not help, escalate to next level

Recovery principles cont.

• Orthogonal to "restart" is "takeover" – service of existing mobile phones are "taken over" by other board after HW failure – ideally phone should not notice
• Method: separate "control" from "data" – all data related to one phone is replicated to one other board
• Efficiency? Can not replicate every time data changes – select "good points" to do replication (transaction concept)
Processes - Generic Servers

- Most processes are "server like"; receive message – do some computation – send reply
- SGSN extends OTP gen_server behaviour:
  - message passing via cast, no reply
  - message passing via call (≈ cast + synchronization + return value)

Example 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.

Improved gen_server

gen_server2:
- handle_call(M,F,A) ->
  apply(M,F,A).

sender:
- Ms = gen_server2:call(Pid, {mobility, attach, [Id]}),
  Ret = gen_server2:call(Pid, {session, activate, [Ms]}),

receiver (file mobility.erl):
  attach(Id) ->
  <do something>.

receiver (file session.erl):
  activate(Ms) ->
  <do something more>.

Example – robust message passing

- Problem: implement "cast" with guaranteed delivery even if receiver crashes before message is handled
- How?
  - Implement cast as: send message + write into persistent storage
  - In receiver: after processing, remove message from storage
  - In startup of receiver (after crash): check for stored messages

Erlang – Concurrency

- "Normal" synchronization primitives, like 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 - Concurrency cont.

- Atomic operations:
  - ets:update_counter()
  - mnesia:transaction()
- "home made" using a transaction handler process (TP)
  - client starts transaction, message to TP
  - client does some "work"
  - client ends transaction, message to TP
  - TP commits "work"
  - "failure" when transaction is started but not ended makes TP revert to state before the start

Erlang - Distribution

- General rule in SGSN: avoid remote communication or synchronization if possible
- Design algorithms that work independently on each board
  - 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, use mnesia or manual
  - many different variants of persistency, redundancy, replication

Example – intra-SGSN routing

- Problem – an incoming signal from a phone is received in the Payload Plane, to which CP should it be routed?
- Old solution: a global resource was used to keep mappings between different "identities" that were linked to the phone and the corresponding CP
- New solution: construct identities in a clever way, encode CP somewhere in Id
- For Ids that are outside SGSN control, send signal to a random CP (rare) or broadcast to all CPs (very rare)

Multicore

- Erlang in theory gives you multicore support "for free"
- The BEAM (Erlang virtual machine) will automatically schedule Erlang processes onto all available cores.
- However – linear speedup is not guaranteed. It is up to the application code to offer enough possibilities for "parallelism". Very easy to get resource bottlenecks.
- Profiling in a multicore environment is hard!
- In SGSN – dual core gave roughly 20% without tuning.

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
  Ex: since "live data" survives in tables/storage, the new version of the code must be able to handle data in both new and old format.

Bugs in Erlang

- Bugs in Erlang / OTP are as common as bugs in SGSN
- How do we protect SGSN against Erlang failures?
- Base: same methods as for SGSN code; recovery by restarts and escalation
- Addition: if restarts local to one Erlang node repeatedly fails to resolve an error condition, then kill that Erlang node
- Using Erlang in a robust way in a distributed system where hardware may suddenly fail is a very hard problem!
Overload Protection

- If CPU load or memory usage goes too 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)

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 "domain specific language".
- To realize this, functional programming is very good!
- But to summarize: FP is a great help – but not vital. Or?

Haskell?

- Could we use Haskell instead of Erlang?
- Not trivial – need to do some fundamental re-design of the system:
  - "one process per mobile phone" – need to implement our own scheduler?
  - "memory protection between processes" – need to separate "data" related to phone #1 from data related to phone #2
  - "recovery from software faults" – how do we crash and restart without losing all data?

Haskell cont.

- Redesign cont.:
  - "concurrency" – sending messages between boards
  - "runtime code replacement" – need to replace broken software without losing the data about the phones
  - "efficiency & memory usage"?
- Reflection: consider Erlang vs. Haskell vs. C++. Which two are the most similar?

Conclusions

Pros:
- Erlang works very well for GPRS traffic control handling
- High level language – concentrate on important parts
- Has the right primitives; fault tolerance, distribution, ...

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