Message-Passing Concurrency in Erlang

ACCU, Oxford, April 14th, 2010

Ulf Wiger
ulf.wiger@erlang-solutions.com
@uwiger
The (original) Problem
Agent-based service...

25-lines switchboard, Natal Province, South Africa 1897
Cross-switchboard calls required human interaction.
Scalability - Resource Partitioning...

SAT’s main telephone exchange, Stockholm 1897 - 7000 lines
"Multiple switchboard" - each operator could work independently.
Automatic Switching - Machine-driven

Ericsson’s 500 Switch 1940s - 500 lines per stage
Stored Program Control - Bugs and all!!!
Consequences of Software Control

- More complex services become possible
  - Call waiting, Call forwarding
  - Call pickup groups, toll-free numbers
  - Conference calls on demand
  - ...

- New hairy problem: Feature interaction

- Higher line density calls for higher reliability

- Language designed by committee, CCITT CHILL (1980), was supposed to address the important problems

- Ericsson designed PLEX, PL163, EriPascal, High-Level PLEX, ...
Digital switching, modular SW design

Ericsson’s AXE10 Switch 1975 - High-Level Language (PLEX)
Ericsson’s PLEX Language

- “Blocks” with signal interfaces
- No shared data
- Fail-fast programming & layered restarts
- Redundant (lock-step) Control Processors

- Very, very proprietary
- Lacks selective message reception
- Very difficult to extend to multi-processor
1958: First phone call completed using SPC technology
LISP

1965: Edsger Dijkstra - the first mutual exclusion algorithm

1970-72: Ericsson drafts the AXE/PLEX design

Niklaus Wirth creates Modula-2

1982: Lamport et al describe The Byzantine Generals Problem

1981-1987: SPOTS Experiments ⇒ Erlang

1991: Erlang publicly announced
The forming of Ericsson CSLab 1980

- Small group of people
  - Bjarne Däcker
  - Göran Båge
  - Seved Torstendahl
  - Mike Williams

- Systematic treatment of Computer Science

- Highly experimental, literally a "laboratory"
Language Experiments

• SPOTS (SPC for POTS)

• Wrote control system in several languages
  • Ada, CHILL, CCS, LPL, Concurrent Euclid, Frames, CLU, OPS4

• Domain experience identified the tricky problems

• Led to yet a new language: Erlang
Properties of Erlang

- **Telecom goodness:**
  - Scalable agent-style concurrency
  - Distribution transparency
  - Fail-fast programming style

- **Managing complexity**
  - Declarative/functional programming inside each process
  - No shared data, loosely coupled components (black-box style design)
  - ”Programming for the correct case”

- **Evolving systems**
  - In-service upgrades
  - (Dynamic typing)
Erlang was never about speed

- Writing software that is
  - Complex
  - Distributed
  - Evolving
  - Fault-tolerant

- However...
Multicore 🤝 Message-passing Concurrency

Erlang/OTP R13B on Tilera Pro 64-core

Big_bang benchmark, 500 processes chatting
- Bound schedulers
- Default (unbound)

(No Tilera-specific optimizations!)

ca 0.4x @ 48 cores
factorial(N) when is_integer(N), N > 0 ->
    N * factorial(N-1);
factorial(0) ->
    1.

area({square, Side}) -> Side * Side;
area({rectangle, B, H}) -> B * H;
area({triangle, B, H}) -> B * H / 2.

- Describe the expected - crash on erroneous input
- Infrastructure handles recovery
-module(my_server).
-export([start_server/2, call/2]).

start_server(F, St0) ->
  spawn_link(fun() ->
    St = F(init, St0),
    server_loop(F, St)
  end).

call(Server, Req) ->
  Ref = erlang:monitor(process, Server),
  Server ! {call, self(), Ref, Req},
  receive
    {Ref, Reply} -> Reply;
    {'DOWN', Ref, _, _, Reason} -> erlang:error(Reason)
  after 5000 ->
    erlang:error(timeout)
  end.
server_loop(F, St) ->
    receive
        {call, From, Ref, Req} ->
            {Reply, NewSt} = F({call, Req}, St),
            From ! {Ref, Reply},
            server_loop(F, NewSt);
        _ -> server_loop(F, St)
    end.
Parameterizing our server

-module(counter).
-export([new/1, inc/2]).

new(InitialValue) ->
    my_server:start_server(fun counter:main/2, InitialValue).

inc(Counter, Value) ->
    my_server:call(Counter, {inc, Value}).

main(init, Initial) ->
    Initial;
main({call, {inc, V}}, N) ->
    N1 = N + V,
    {N1, N1}.
Running it from the interactive shell

```erlang
Eshell V5.7.2  (abort with ^G)
1> c(my_server).
   {ok,my_server}
2> c(counter).
   {ok,counter}
3> C = counter:new(0).
   <0.44.0>
4> counter:inc(C,1).
   1
5> counter:inc(C,5).
   6
6> counter:inc(C,-2).
   4
7> counter:inc(C,foo).
   =ERROR REPORT==== 6-Nov-2009::08:23:21 ===
   Error in process <0.44.0> with exit value: ...
   ** exception exit: badarith
      in function  counter:counter/2
      in call from my_server:server_loop/2
```
Scalability in the Cloud?

- You just saw it!
- Lightweight processes
- Distribution transparency
- Asynchronous message passing
- Monitoring and recovery/re-routing

```
1. register(Name, self())
2. whereis(Name)
3. P ! Msg
```
The “Always Copy” Illusion

- Conceptually, messages are always copied
- A necessity in the distributed case
- Under the hood, data may be passed by reference
  - “copy on write”
- Per-process garbage collection
- Transparent to the program
- No explicit sharing!
Erlang as (Distributed) System Glue

A “port” to the outside world looks like a process to Erlang

- C code as a linked-in driver
- Port process, via pipe or socket
- C code wrapped as an Erlang module

Copyright 2008 - Erlang Training and Consulting Ltd
Robust systems can be built using layering.

Program for the correct case.
Handling sockets in Erlang

1. Static process opens listen socket
2. Spawns an acceptor process
3. Acceptor receives incoming
4. Acks back to socket owner
5. New acceptor is spawned
6. Replies sent directly to socket
Middle-man Processes

```
spawn_link(PidA, PidB) ->
    spawn_link(fun() ->
        loop(#state{a_pid = PidA, b_pid = PidB})
    end).
```

```
await_negotiation(State) ->
    receive
        {From, {simple_xml, [{"offer", Attrs, Content}]}} ->
            HisOffer =
                inspect_offer(Attrs, Content),
            Offer = calc_offer(HisOffer, State),
            From ! {self(), Offer};
    ...end.
```

- Practical because of light-weight concurrency
- Normalizes messages
- Main process can pattern-match on messages
- Keeps the main logic clear
Erlang Bends Your Mind...

- Processes are cheap and plentiful!
  - When you need a process - just create one!
  - Don’t ration processes - use exactly as many as you need
  - No need for thread pools - reusing processes is really a pain!

- Message-passing is cheap!
  - Use processes to separate concerns
  - Middle-man processes useful for transforming data

- Processes can monitor each other
  - Enables out-of-band error handling

- Use Concurrency as a Modelling Paradigm!
Example: RFC 3588 - DIAMETER Base Protocol

<table>
<thead>
<tr>
<th>state</th>
<th>event</th>
<th>action</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-Open</td>
<td>Send-Message</td>
<td>I-Snd-Message</td>
<td>I-Open</td>
</tr>
<tr>
<td>I-Rcv-Message</td>
<td>Process</td>
<td>I-Open</td>
<td></td>
</tr>
<tr>
<td>I-Rcv-DWR</td>
<td>Process-DWR,</td>
<td>I-Snd-DWA</td>
<td></td>
</tr>
<tr>
<td>I-Rcv-DWA</td>
<td>Process-DWA</td>
<td>I-Open</td>
<td></td>
</tr>
<tr>
<td>R-Conn-CER</td>
<td>R-Reject</td>
<td>I-Open</td>
<td></td>
</tr>
<tr>
<td>Stop</td>
<td>I-Snd-DPR</td>
<td>Closing</td>
<td></td>
</tr>
</tbody>
</table>

- Three state machines described as one
- Implies a single-threaded event loop
- Introduces accidental complexity
Use processes to separate concerns

Handshake FSM
- Capabilities exchange
- Leader election
- Only active during handshake

Dynamic request handler
- One per request

Transport FSM
- Handles heartbeat logic (RFC 3539)

Service FSM
- Request routing
- Failover
- Retransmission
Closing words

- Poor concurrency models can lead to complexity explosion
- Much accidental complexity is often viewed as a given
- Event-based programming is the new GOTO

- Message passing Concurrency is a powerful structuring model
- Fault handling is an oft overlooked aspect of Erlang-style Concurrency

Photos from http://www.ericssonhistory.com

Copyright 2008 - Erlang Training and Consulting Ltd