Erlang Training and Consulting Ltd

Message-Passing Concurrency in Erlang

ACCU, Oxford, April 14th, 2010

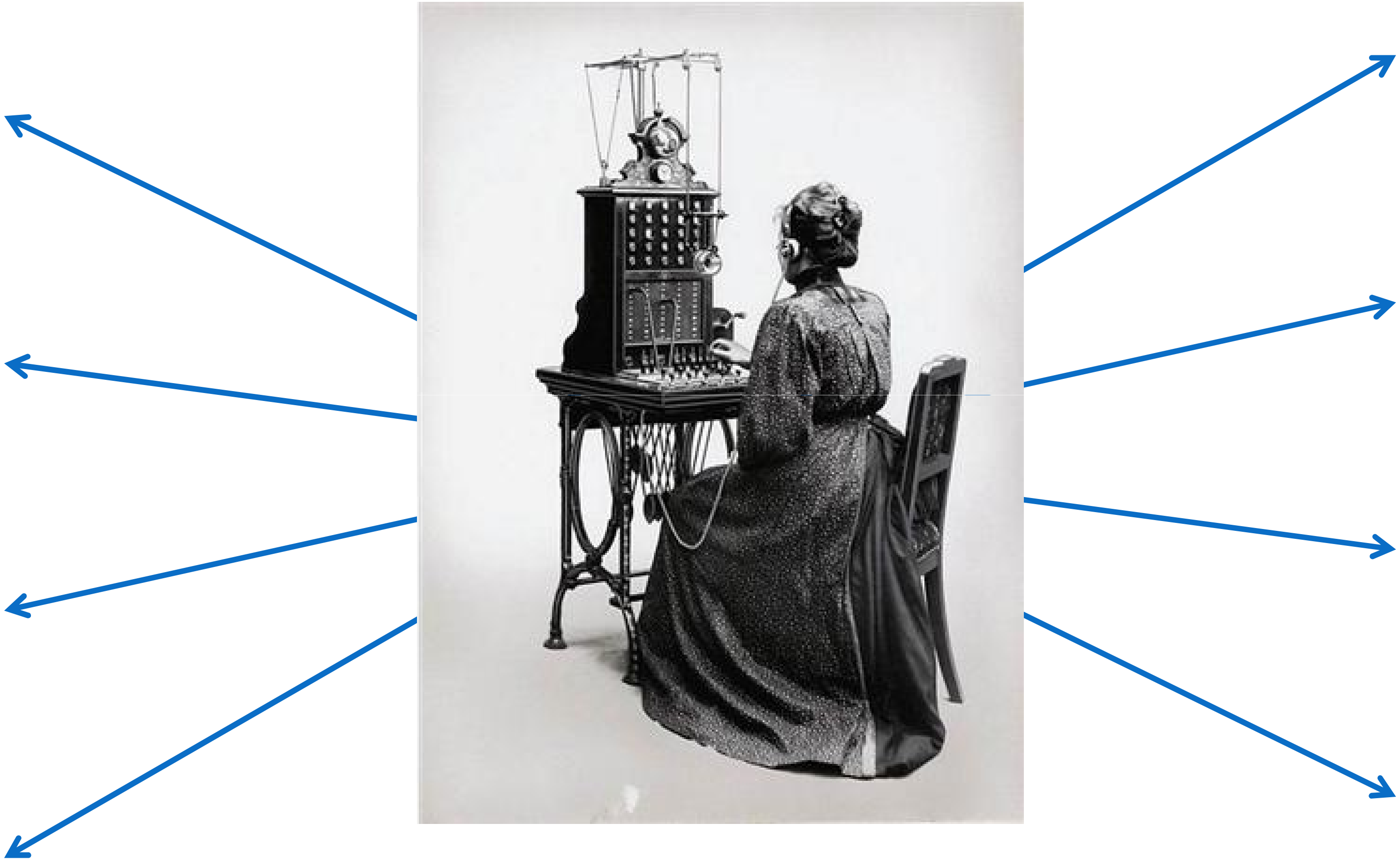
Ulf Wiger

ulf.wiger@erlang-solutions.com

@uwiger

The (original) Problem

Agent-based service...

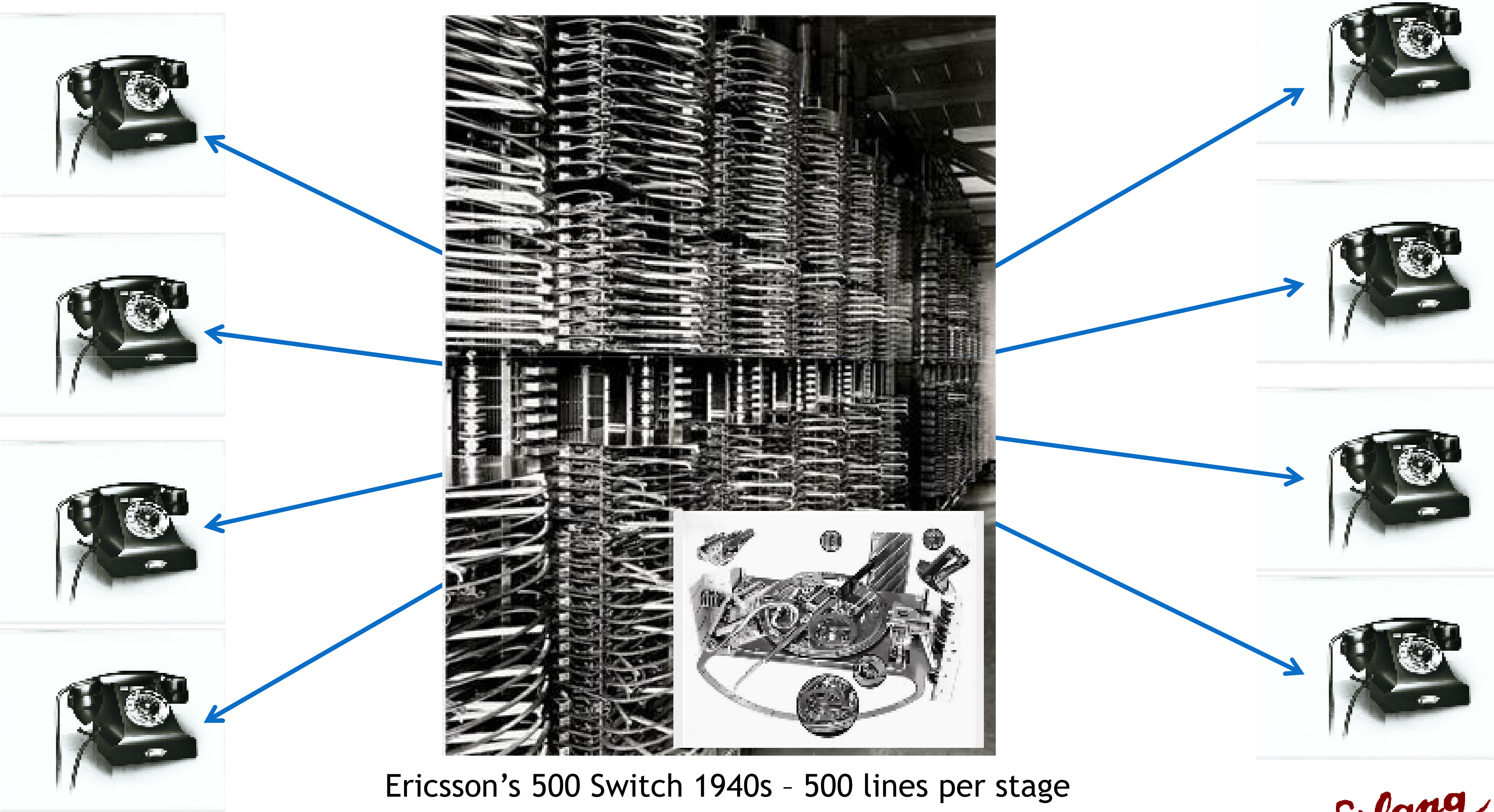


25-lines switchboard, Natal Province, South Africa 1897
Cross-switchboard calls required human interaction.

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

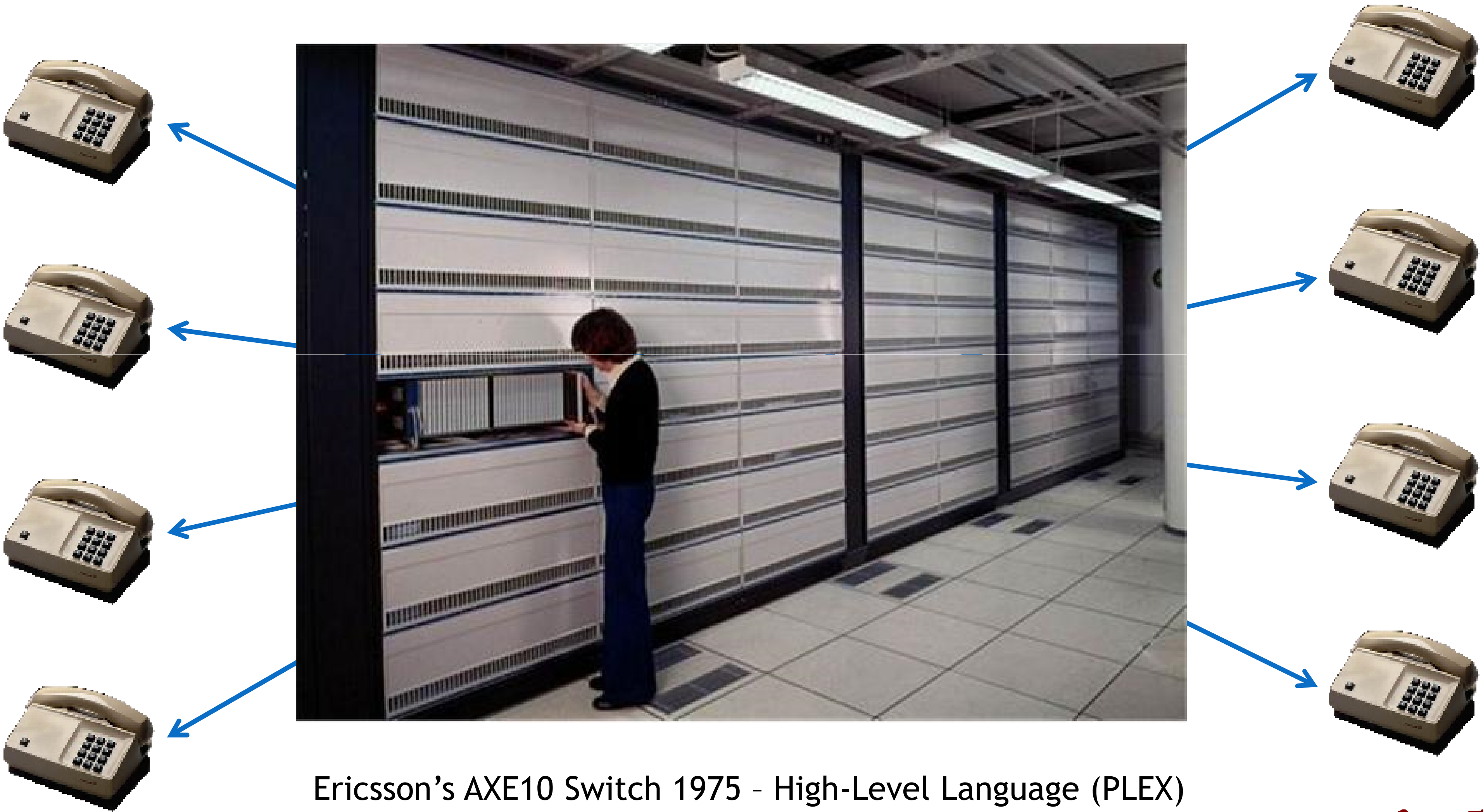
Stored Program Control – Bugs and all!!!

Ericsson's AKE12 Switch 1968 –
Computerized Electromagnetic code switching

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

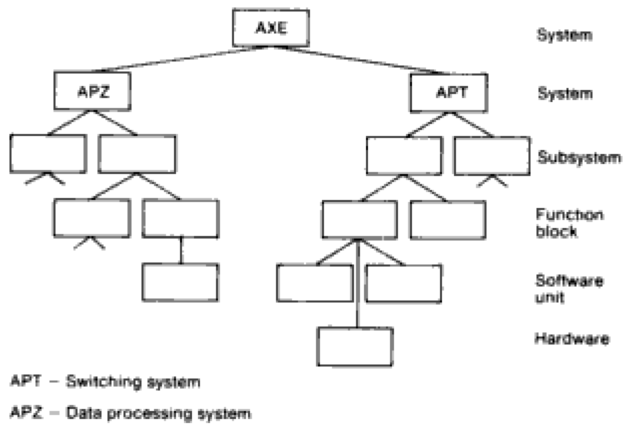


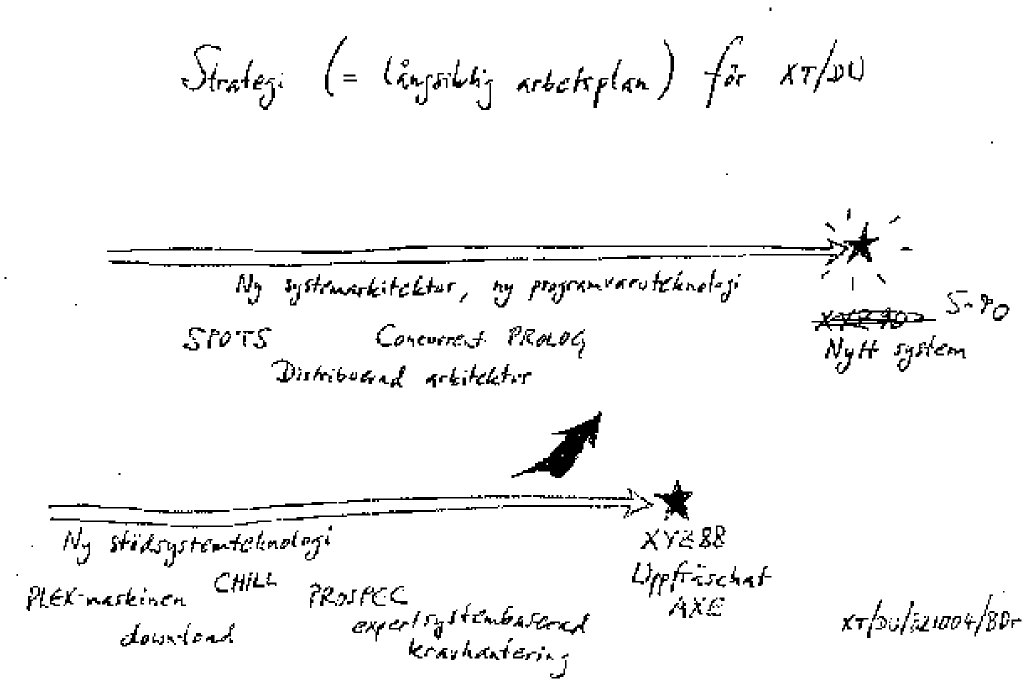
Fig. 3. AXE System Structure Levels

- Very, very proprietary
- Lacks selective message reception
- Very difficult to extend to multi-processor

Checkpoint

- **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**
- 1978: C.A.R. Hoare publishes CSP book
  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



CSLab plan 1981 (Bjarne Däcker)

- Small group of people
  - Bjarne Däcker
  - Göran Båge
  - Seved Torstendahl
  - Mike Williams
- Systematic treatment of Computer Science
- Highly experimental, literally a ”laboratory”

Language Experiments

http://video.google.com/videoplay?docid=-5830318882717959520#

- 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**

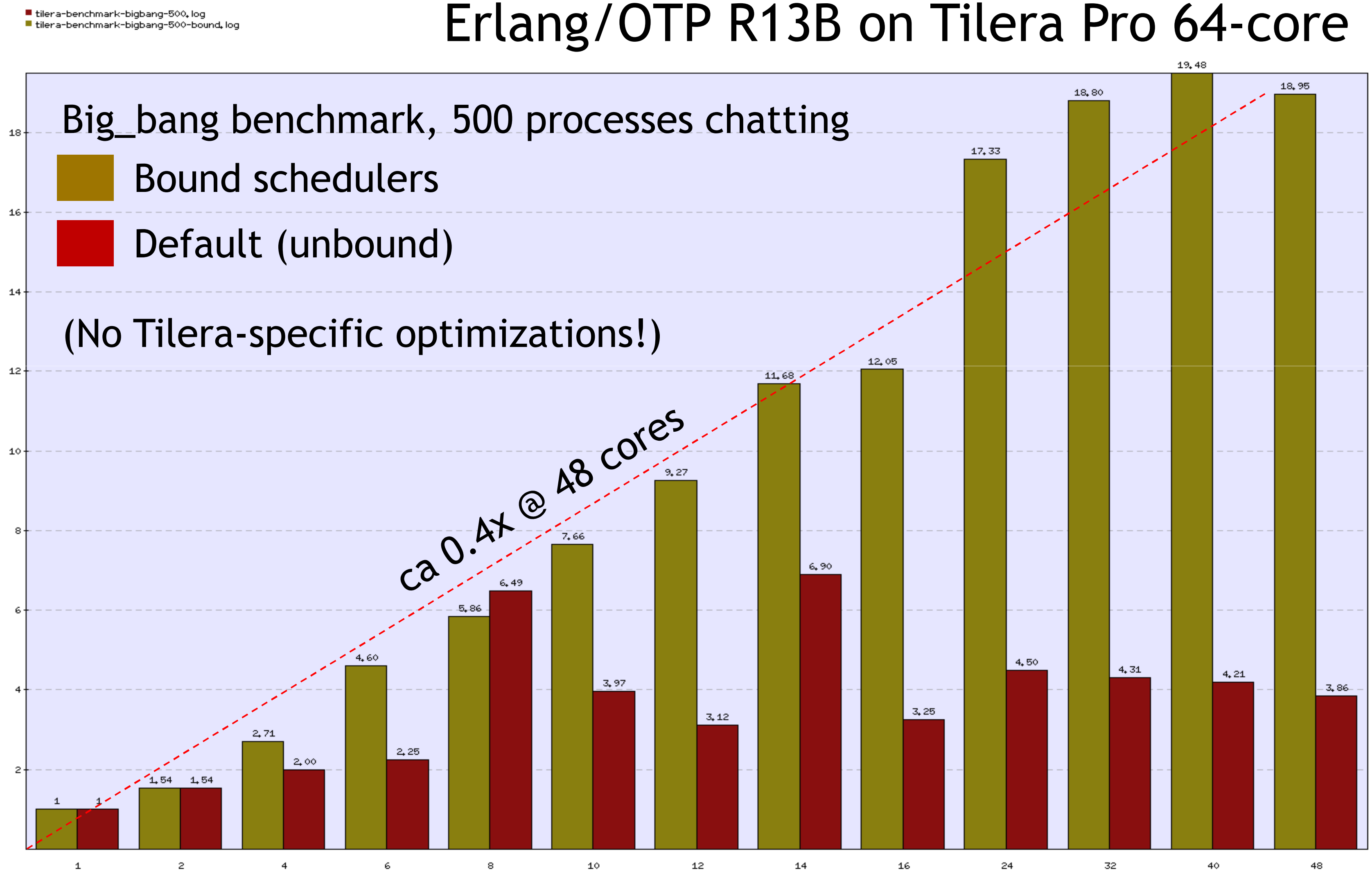
Properties of Erlang

- **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



Program for the correct case - Patterns

```
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

Erlang Concurrency

```
-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.
```

Observation: **From** can refer to a remote process (distribution transparency)

Selective receive

Erlang Concurrency, cont...

```
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

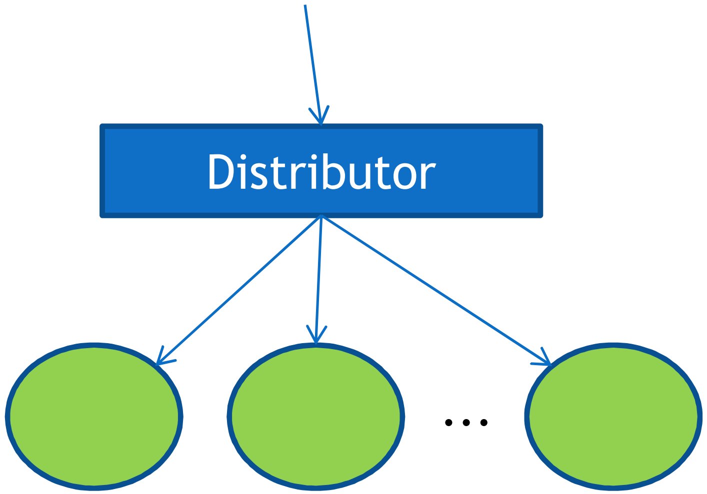
```
-module(counter).
-export([new/1, inc/2]).

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}.
```

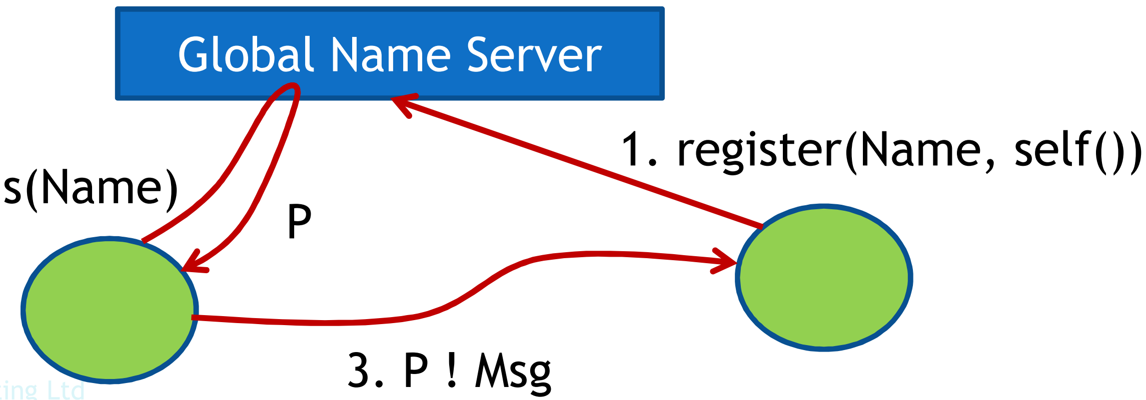
Running it from the interactive shell

```
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
6> counter:inc(C,5).
6
7> counter:inc(C,-2).
4
8> 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



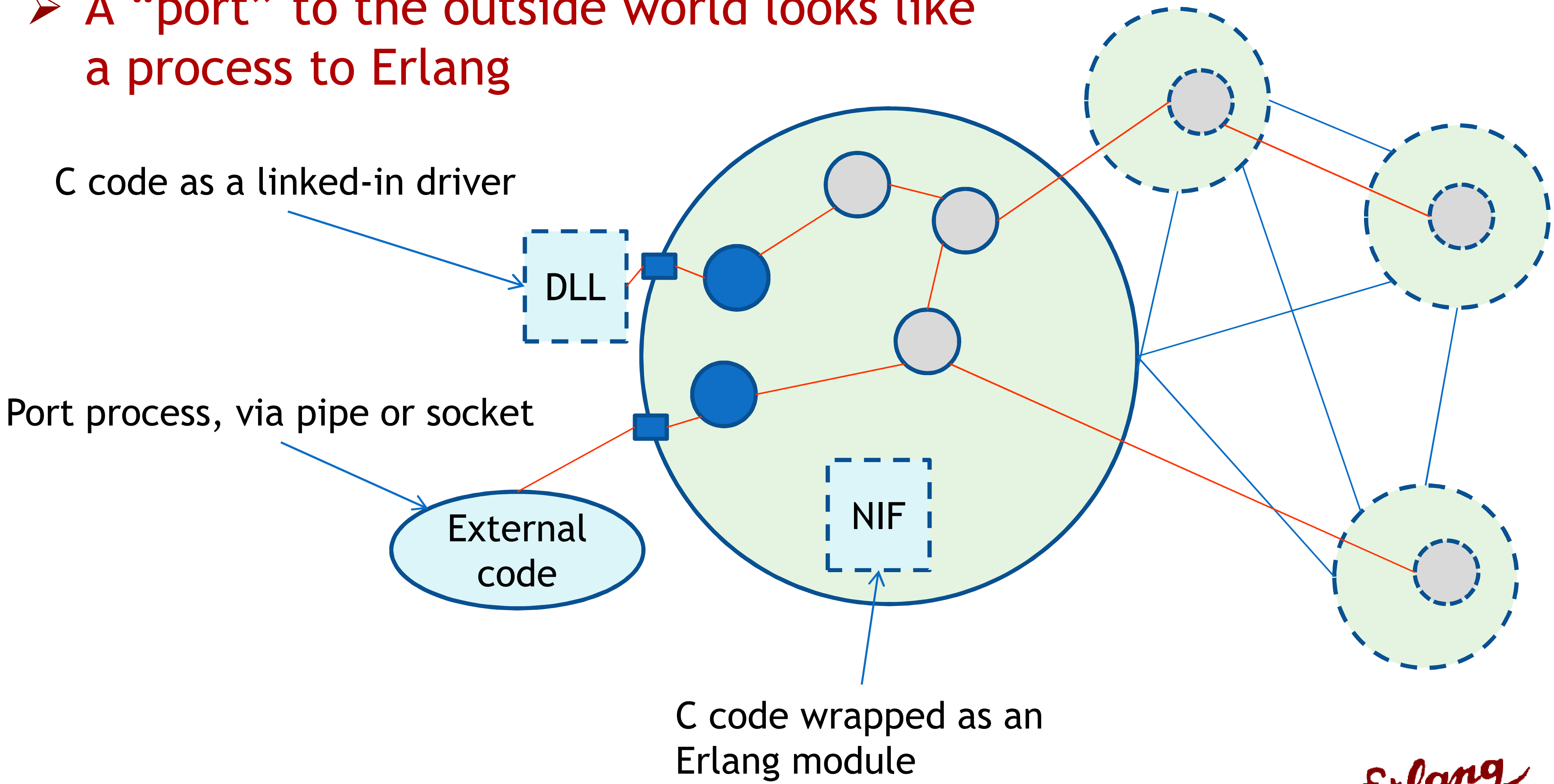


The “Always Copy” Illusion

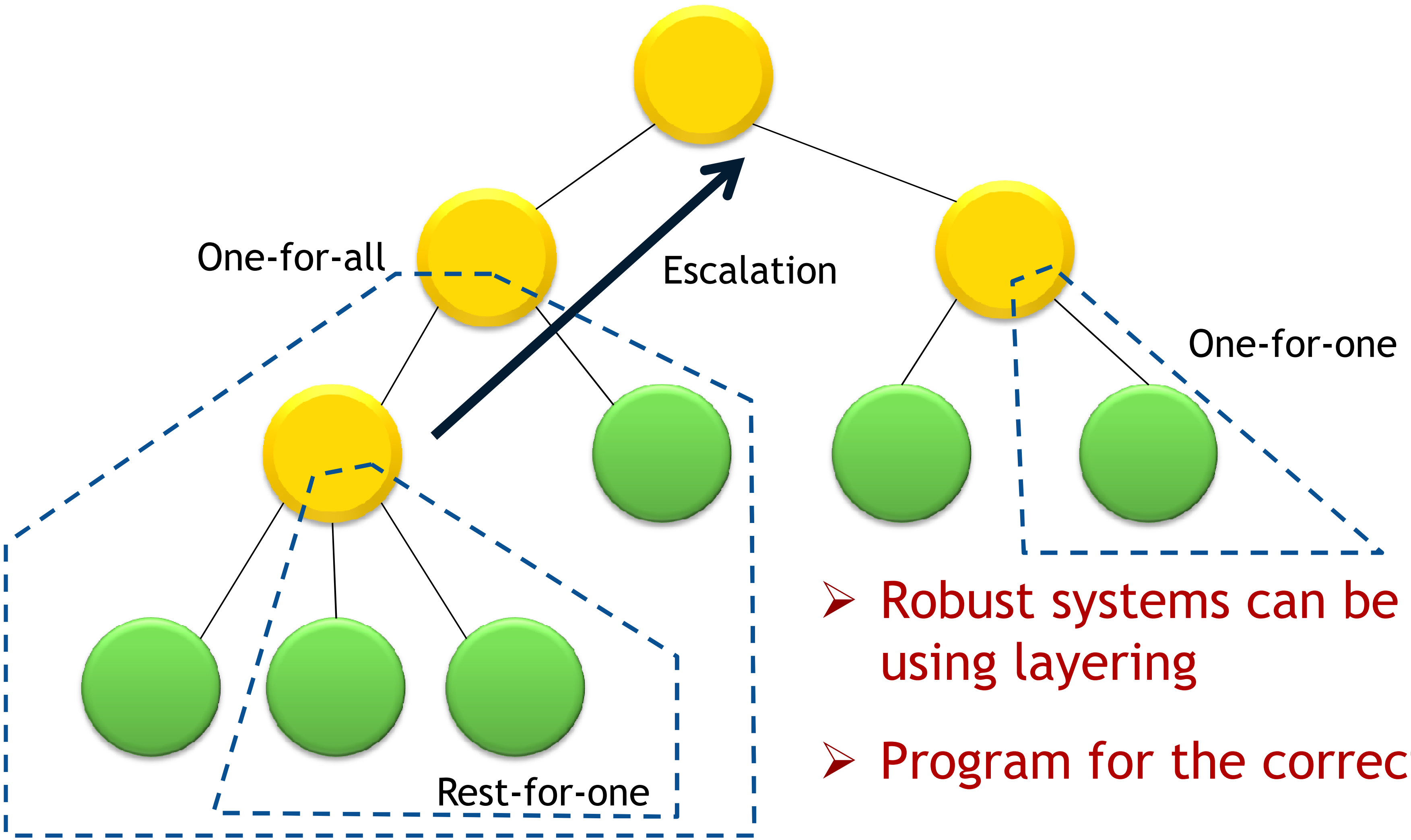
- Conceptually, messages are *always* copied
- A necessity in the distributed case
- Under the hood, data may be passed by reference
  - “copy on write”
- Per-process garbage collection
- Transparent to the program
- No explicit sharing!

Erlang as (Distributed) System Glue

- A "port" to the outside world looks like a process to Erlang

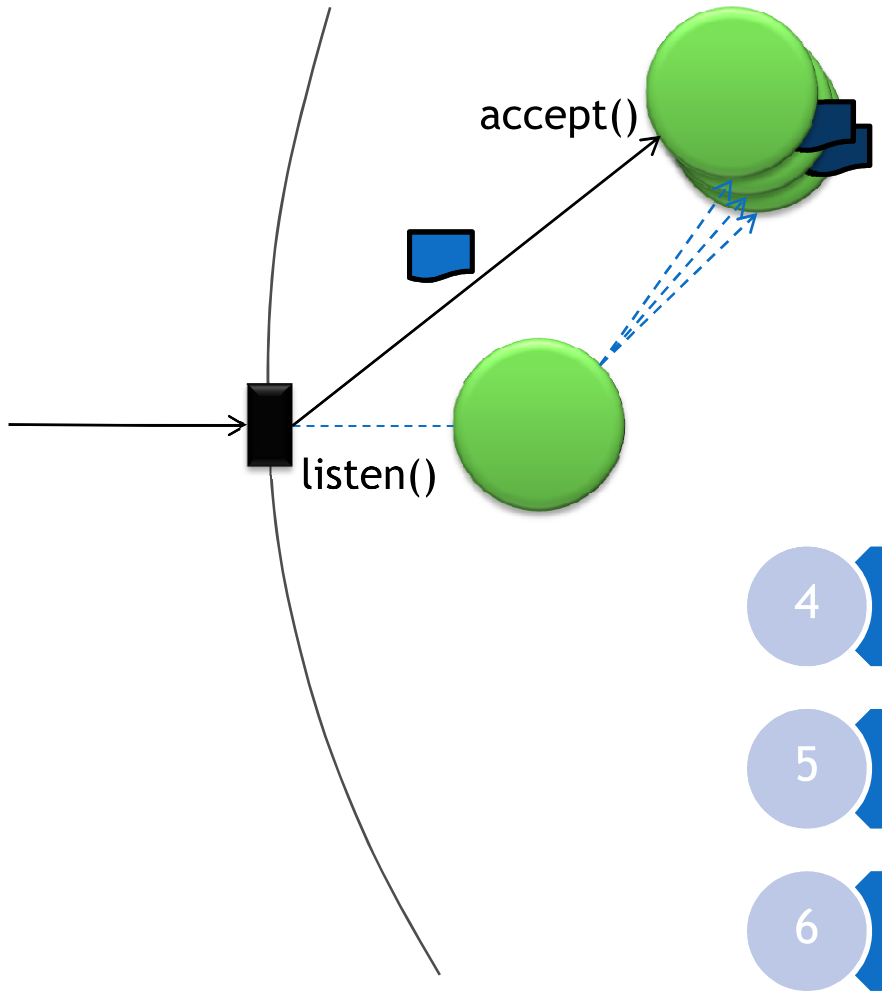


Supervisors – Out-of-Band Error Handling



- Robust systems can be built using layering
- Program for the correct case

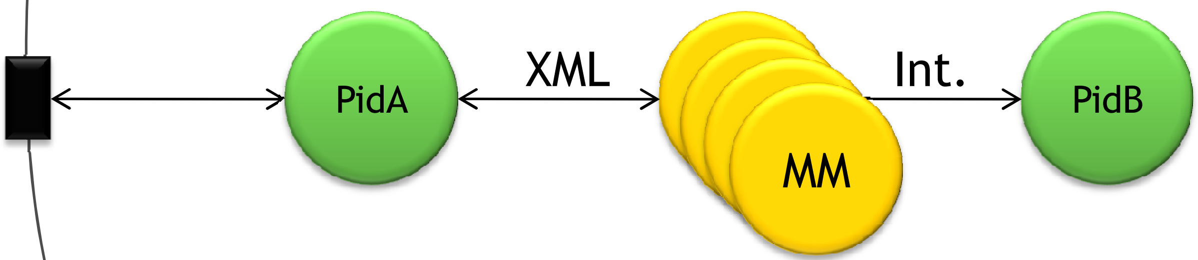
Handling sockets in Erlang



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).
```




```
loop(#state{a_pid = PidA, b_pid = PidB} = State) ->
    receive
       {PidA, MsgBin} when is_binary(MsgBin) ->
           {simple_xml, _} = Msg = vccXml:simple_xml(MsgBin),
           PidB ! {self(), Msg},
           loop(State);
       {PidB, {simple_xml, _} = Msg} ->
           Bin = vccXml:to_XML(Msg),
           PidA ! {self(), Bin},
           loop(State)
    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!

Language Model Affects our Thinking

Example: RFC 3588 – DIAMETER Base Protocol

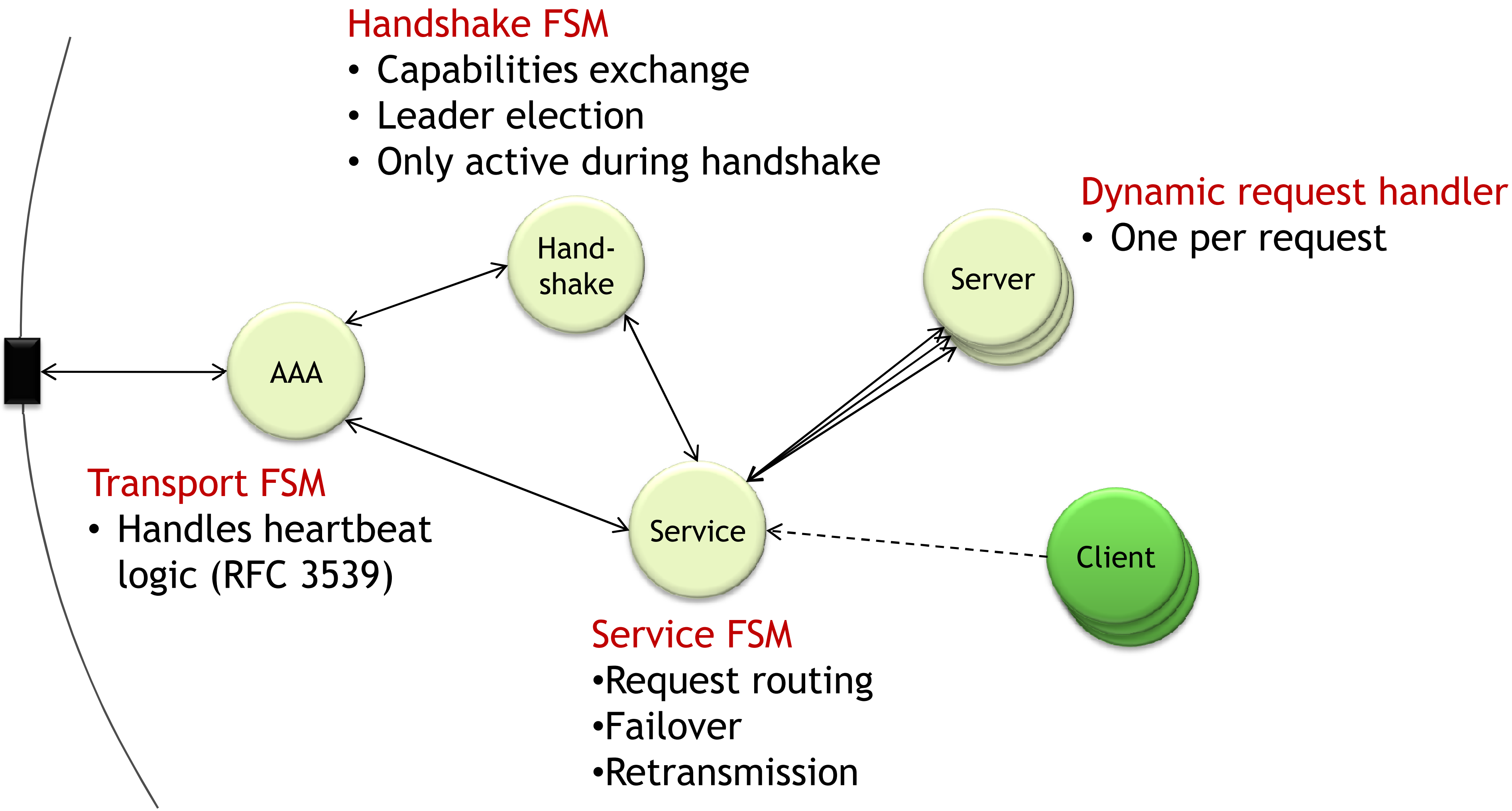
```
state            event              action         next state
-----------------------------------------------------------------
...
I-Open           Send-Message       I-Snd-Message  I-Open
                 I-Rcv-Message      Process        I-Open
                 I-Rcv-DWR          Process-DWR,   I-Open
                                    I-Snd-DWA
                 I-Rcv-DWA          Process-DWA    I-Open
                 R-Conn-CER         R-Reject       I-Open
                 Stop               I-Snd-DPR      Closing
...
```

Transport FSM

Handshake FSM

- Three state machines described as one
- Implies a single-threaded event loop
- Introduces accidental complexity

Use processes to separate concerns



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