Erlang Solutions Ltd

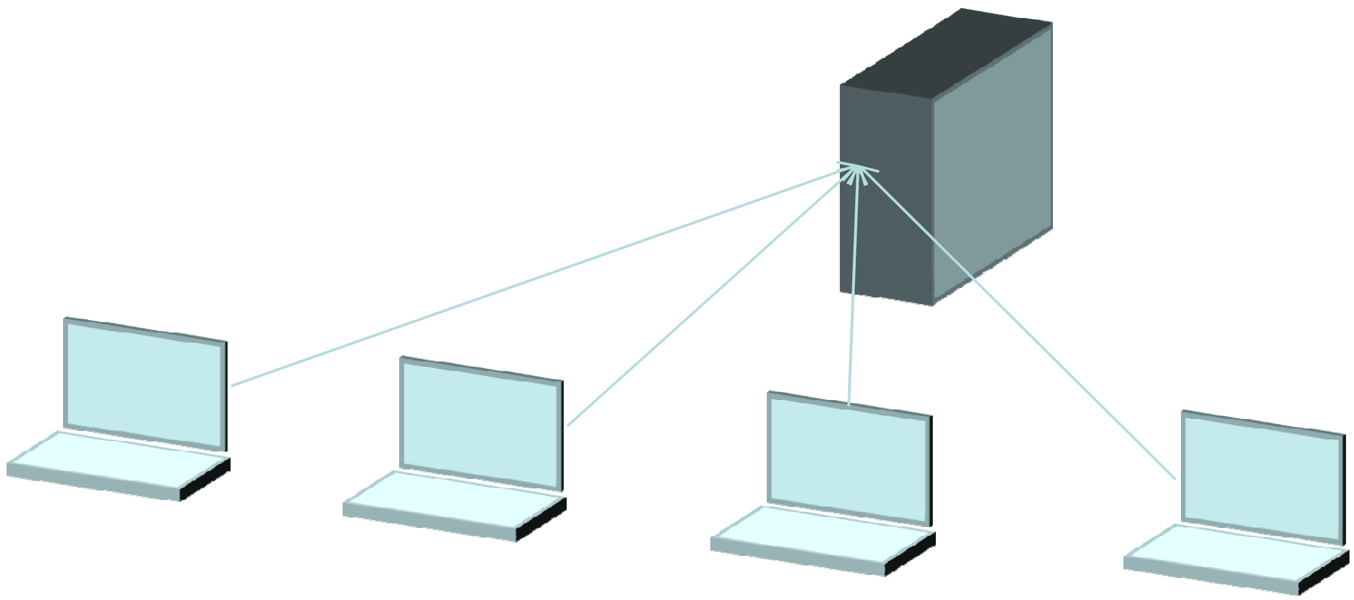
Death by Accidental Complexity

QCon
London, March 12, 2010
Ulf Wiger, CTO Erlang Solutions Ltd.

Ulf Wiger
ulf.wiger@erlang-solutions.com

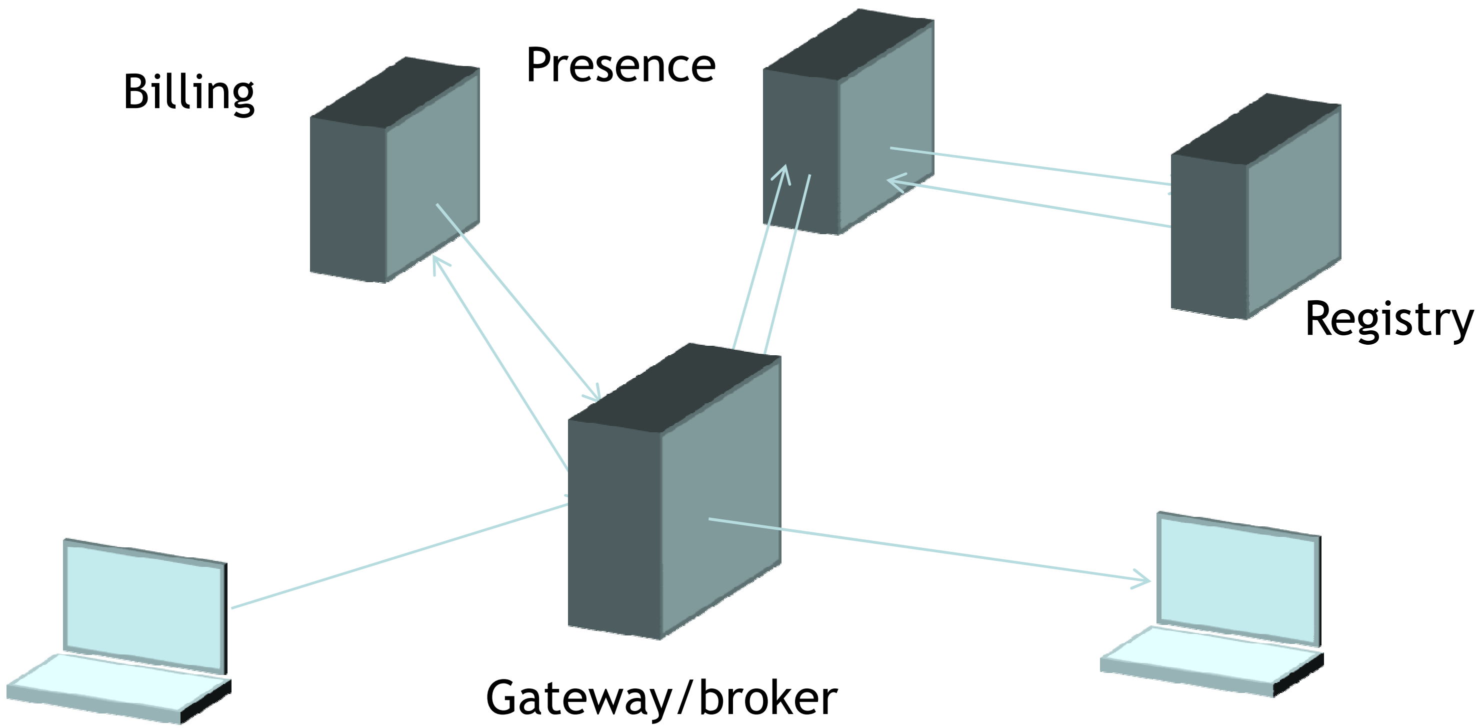
Brain-dead concurrency

- A.k.a. Embarrassingly Parallel
- E.g. Plain web server
  - Minimal dependencies, request/reply pattern

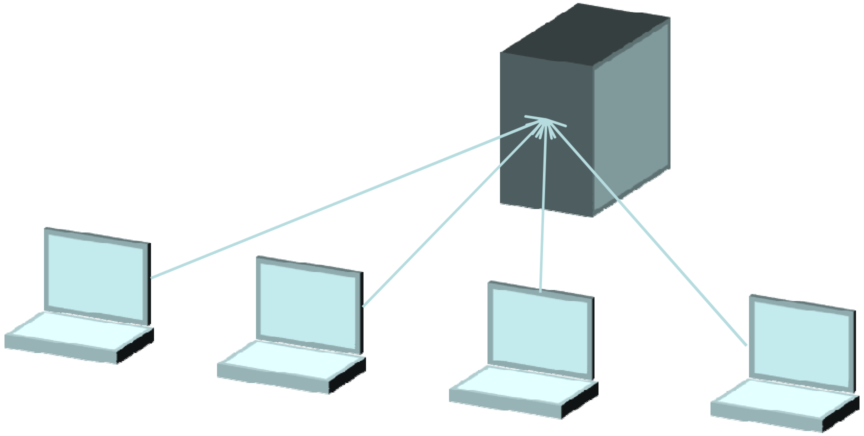


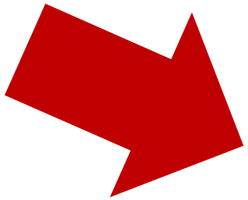
Mind-blowing concurrency

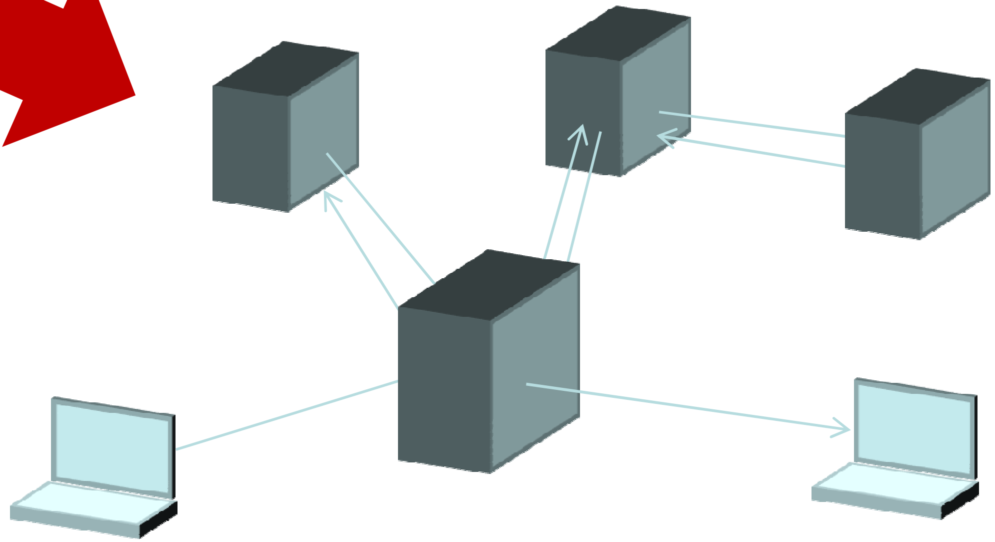
- Complex, multi-way signalling
- Multiple failure domains



Natural product evolution



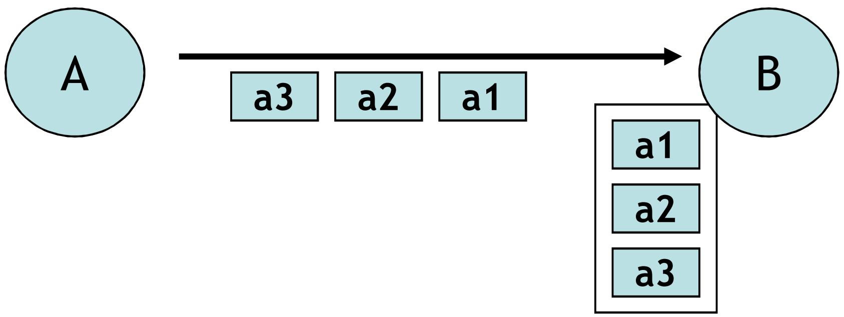




- From simple to more complex
- Structuring and encapsulation should allow us to support this

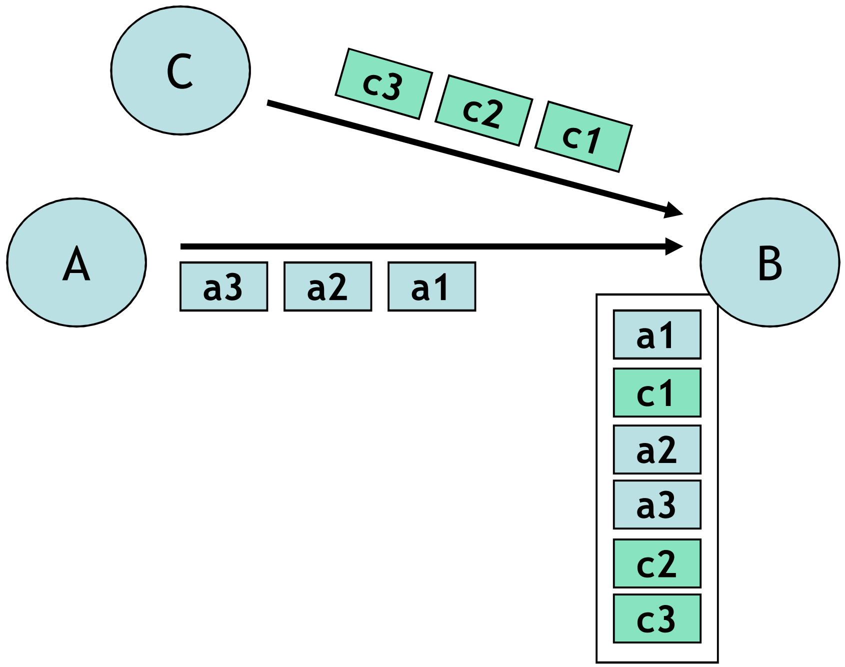
Message ordering (1)

- All messages from A to B
  must arrive in the same order as they were sent.



Message ordering (2)

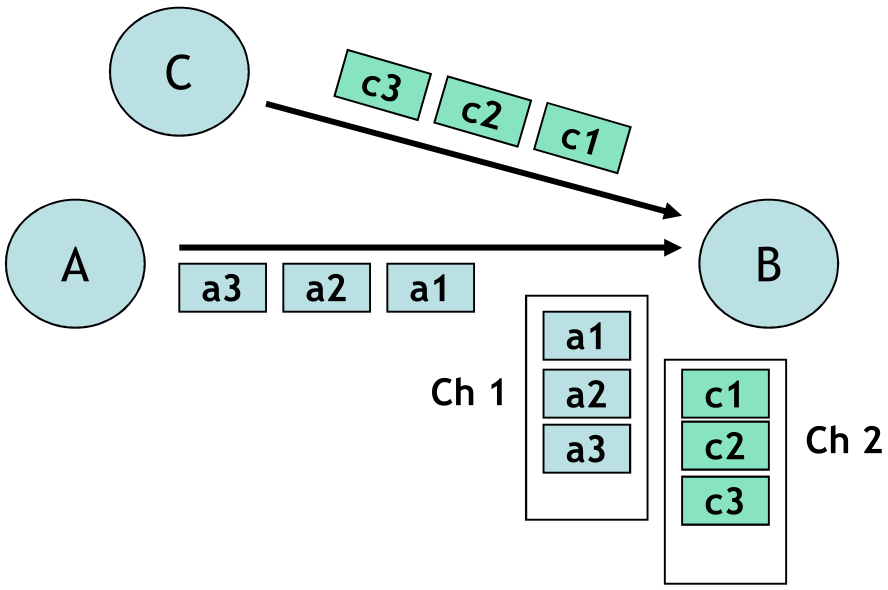
- Oft forgotten rule:
  - The language should not force a specific global order upon the programmer
- Common (naïve) implementation:
  - Event loop dispatches in order of arrival
  - Easy to implement
  - But forces the programmer to deal with all possible orderings of events



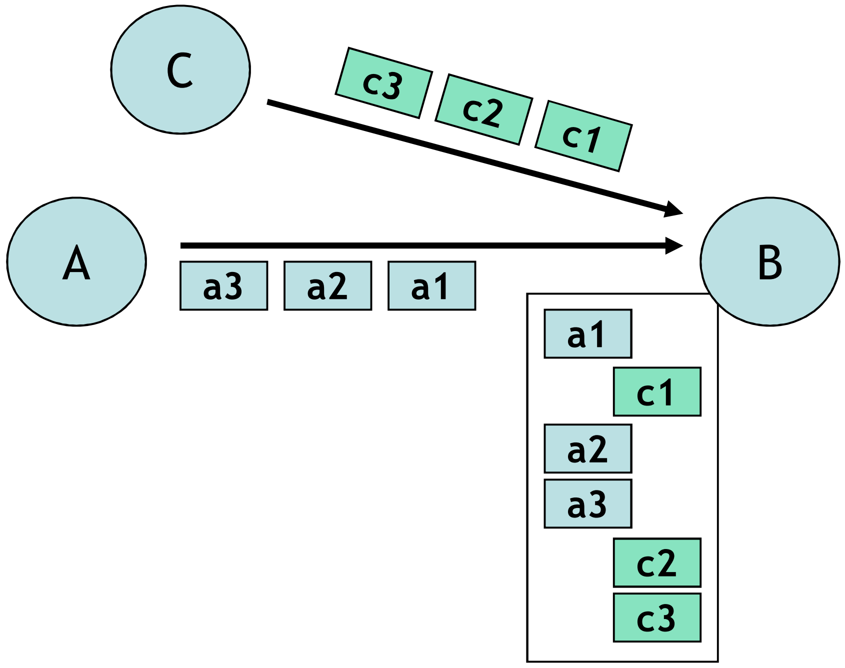
Erlang

Selective Message Reception

- Separate message queues
  - Ability to select msgs out of order



- One queue per process
  - + ability to select msgs out of order



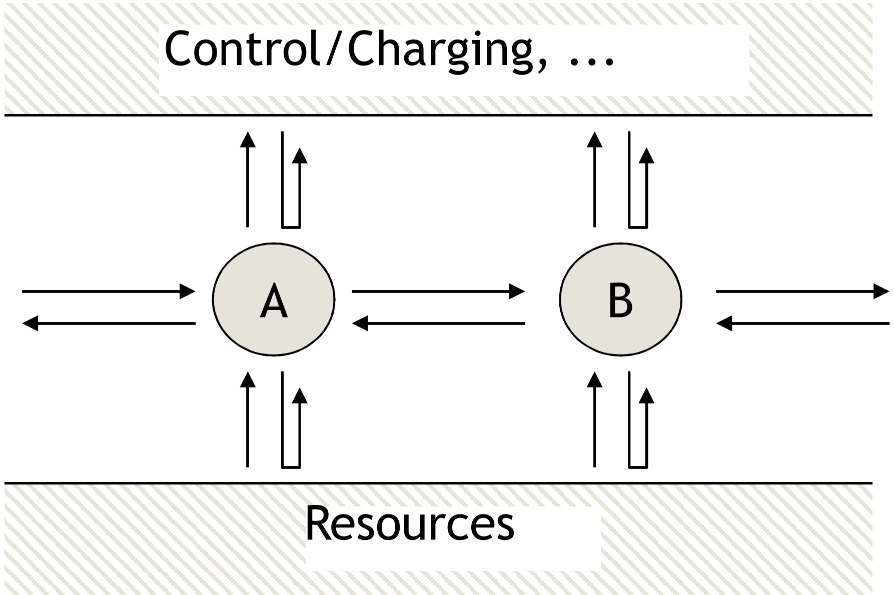
Session Broker Scenario

One or more stateful processes

Interaction with multiple uncoordinated message sources

Message sequences may (and invariably will) interleave

Telecom "Half-Call" model



A = originating side
B = terminating side

Programming Experiment

Demo system used in Ericsson's Introductory Erlang Course

- assignment: write a control program for a POTS subscriber loop

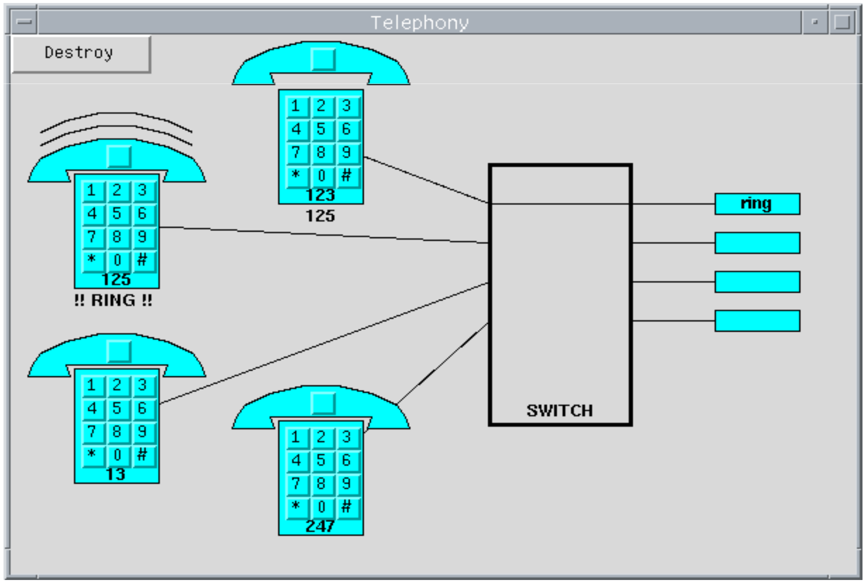
We will re-write the control loop using different semantics.

- Selective message passing
- Event dispatch

Note well: no error handling (usually the most complex part)

http://github.com/uwiger/pots

"POTS": Plain Ordinary Telephony System – Trivial schoolbook example of telephony (as simple as it gets)



Demo...

Erlang – two approaches

Event-based programming

```
loop(Module, S) ->
    receive
        Msg ->   % matches any message
            S1 = Module:event(Msg, S),
            loop(Module, S1)
    end.
```

Selective receive FSM

```
call(Server, Request) ->
    Server ! {call, self(), Request},
    receive
        {Server, reply, Reply} ->
            Reply
    after 5000 ->   % milliseconds
        exit(timeout)
    end.
```

We can use Erlang syntax to illustrate both models (hooray!!)

POTS Control Loop - Original Impl. (1/3)

```
start() -> idle().

idle() ->
  receive
     {lim, offhook} ->
         lim:start_tone(dial),
         getting_first_
     {lim, {digi
         idle();
     {hc, {reque
         Pid ! {
         lim:sta
         ringing
     Other ->
         io:form
         idle()
    end.
```

inline selective receive

Synchronous HW control

```
start_tone(Tone)->
  call({start_tone, Tone}).

call(Request) ->
  Ref = make_ref(),
  lim ! {request, Request, Ref, self()},
  receive
    {lim, Ref, {_ReplyTag, Reply}} ->
          Reply
  end.
```

POTS Control Loop - Original Impl. (2/3)

```
getting_first_digit() ->
   receive
      {lim, onhook} ->
         lim:stop_tone(),
         idle();
      {lim, {digit, Digit}} ->
         lim:stop_tone(),
         getting_number(Digit,
             number:analyse(Digit, number:valid_sequences()));
      {hc, {request_connection, Pid}} ->
         Pid ! {hc, {reject, self()}},
         getting_first_digit();
      _Other ->  % unknown message - ignore
         getting_first_digit()
   end.
```

POTS Control Loop - Original Impl. (3/3)

```
calling_B(PidB) ->
    receive
        {lim, onhook} ->
            idle();
        {lim, {digit, _Digit}} ->
            calling_B(PidB);
        {hc, {accept, PidB}} ->
            lim:start_tone(ring),
            ringing_A_side(PidB);
        {hc, {reject, PidB}} ->
            lim:start_tone(busy),
            wait_on_hook(true);
        {hc, {request_connection, Pid}} ->
            Pid ! {hc, {reject, self()}},
            calling_B(PidB);
        _Other ->  % unknown message - ignore
            calling_B(PidB)
    end.
```

Experiment:
Rewrite the program using an event-based model

Event-based vsn, blocking HW control (1/3)

```
%% simple main event loop with FIFO semantics
event_loop(M, S) ->
   case (receive Msg -> Msg end) of
      {From, Event} ->
          dispatch(From, Event, M, S);
      {From, Ref, Event} ->
          dispatch(From, Event, M, S);
      Other ->
          io:format(”Unknown msg: ~p~n", [Other]),
          exit({unknown_msg, Other})
    end.

dispatch(From, Event, M, S) when atom(Event) ->
   {ok, NewState} = M:Event(From, S),
   event_loop(M, NewState);
dispatch(From, {Event, Arg}, M, S) ->
   {ok, NewState} = M:Event(From, Arg, S),
   event_loop(M, NewState).
```

Event-based vsn, blocking HW control (2/3)

```
offhook(lim, #s{state = idle} = S) ->
   lim:start_tone(dial),
   {ok, S#s{state = getting_first_digit}};
offhook(lim, #s{state = {ringing_B_side, PidA}} = S) ->
   lim:stop_ringing(),
   PidA ! {hc, {connect, self()}},
   {ok, S#s{state = {speech, PidA}}};
offhook(From, S) ->
   io:format(”Unknown message in ~p: ~p~n",
             [S#s.state, {From, offhook}]),
   {ok, S}.
```

Synchronous HW control (callout pointing to `lim:stop_ringing()`)

Event-based vsn, blocking HW control (3/3)

```
onhook(lim, #s{state = getting_first_digit} = S) ->
   lim:stop_tone(),
   {ok, S#s{state = idle}};
onhook(lim,#s{state={getting_number,{_Num,_Valid}}} = S) ->
   {ok, S#s{state = idle}};
onhook(lim, #s{state = {calling_B, _PidB}} = S) ->
   {ok, S#s{state = idle}};
onhook(lim, #s{state = {ringing_A_side, PidB}} = S) ->
   PidB ! {hc, {cancel, self()}},
   lim:stop_tone(),
   {ok, S#s{state = idle}};
onhook(lim, #s{state = {speech, OtherPid}} = S) ->
   lim:disconnect_from(OtherPid),
   OtherPid ! {hc, {cancel, self()}},
   {ok, S#s{state = idle}};
...
```

A bit awkward
(FSM programming ”inside-out”),
but manageable.

Add the non-blocking restriction

(first, naive, implementation)

Non-blocking, event-based (1/3)

Asynchronous HW control

```
offhook(lim, #s{state = idle} = S) ->
   lim_asynch:start_tone(dial),
   {ok, S#s{state = {{await_tone_start, dial},
                     getting_first_digit}}};
offhook(lim, #s{state = {ringing_B_side, PidA}} = S) ->
   lim_asynch:stop_ringing(),
   PidA ! {hc, {connect, self()}},
   {ok, S#s{state = {await_ringing_stop, {speech, PidA}}}};
offhook(lim, S) ->
   io:format("Got unknown message in ~p: ~p~n",
       [S#s.state, {lim, offhook}]),
   {ok, S}.
```

Non-blocking, event-based (2/3)

```
digit(lim, Digit, #s{state = getting_first_digit} = S) ->
    %% lim:stop_tone(),
    %% idle();
    %% CHALLENGE: Since stop_tone() is no longer a synchronous
    %% operation, continuing with number analysis is no longer
    %% straightforward. We can either continue and somehow log that
    %% we are waiting for a message, or we enter the state await_tone_stop
    %% and note that we have more processing to do. The former approach
    %% would get us into trouble if an invalid digit is pressed, since
    %% we then need to start a fault tone. The latter approach seems more
    %% clear and consistent. NOTE: we must remember to also write
    %% corresponding code in stop_tone_reply().
    lim_asynch:stop_tone(),
    {ok, S#s{state = {await_tone_stop,
                      {continue, fun(S1) ->
                                         f_first_digit(Digit, S1)
                                 end}}}};
```

Non-blocking, event-based (3/3)

```
start_tone_reply(lim, {Type, yes},
   #s{state = {{await_tone_start, Type}, NextState}} = S) ->
   {ok, S#s{state = NextState}}.

stop_tone_reply(lim,_,#s{state={await_tone_stop,Next}} =S) ->
   %% CHALLENGE: Must remember to check NextState. An alternative would
   %% be to always perform this check on return, but this would increase
   %% the overhead and increase the risk of entering infinite loops.
   case NextState of
      {continue, Cont} when function(Cont) ->
          Cont(S#s{state = Next});
      _ ->
          {ok, S#s{state = Next}}
   end.
```

Quite tricky, but the program still isn't timing-safe. (Demo...)

Global State-Event Matrix

FIFO semantics, asynchronous hardware API

	idle	getting first digit	getting number	calling B	ringing A-side	speech	ringing B-side	wait on-hook	await tone start	await tone stop	await ringing start	await ringing stop	await pid with telnr	await conn-ect	await dis-connect
offhook	O	X	X	X	X	X	O	X	X	X	D	X	X	X	X
onhook	X	O	O	O	O	O	O	O	D	D	D	D	D	D	D
digit	—	O	O	—	—	—	—	—	D	D	D	D	D	D	—
connect	—	—	—	—	O	—	—	—	D	X	X	X	X	X	X
request connection	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
reject	—	—	—	O	—	—	—	—	X	X	X	X	X	X	X
accept	—	—	—	O	—	—	—	—	X	X	X	X	X	X	X
cancel	—	—	—	—	—	—	—	—	X	D	D	D	X	D	X
start tone reply	X	X	X	X	X	X	X	X	O	X	X	X	X	X	X
stop tone reply	X	X	X	X	X	X	X	X	X	O	X	X	X	X	X
start ringing reply	X	X	X	X	X	X	X	X	X	X	O	X	X	X	X
stop ringing reply	X	X	X	X	X	X	X	X	X	X	X	O	X	X	X
pid with telnr reply	X	X	X	X	X	X	X	X	X	X	X	X	O	X	X
connect reply	X	X	X	X	X	X	X	X	X	X	X	X	X	O	X
disconnect reply	X	X	X	X	X	X	X	X	X	X	X	X	X	X	O

Apparent Problems

- The whole matrix needs to be revisited if messages/features are added or removed.
- What to do in each cell is by no means obvious – depends on history.
- What to do when an unexpected message arrives in a transition state is practically never specified (we must invent some reasonable response.)
- Abstraction is broken, encapsulation is broken
- Code reuse becomes practically impossible

Non-blocking, using message filter (1/2)

```
digit(lim, Digit, #s{state = getting_first_digit} = S) ->
   %% CHALLENGE: ...<same as before>
   Ref = lim_asynch:stop_tone(),
   {ok, S#s{state = {await_tone_stop,
                     {continue, fun(S1) ->
                                    f_first_digit(Digit, S1)
                                end}}},
    #recv{lim = Ref, _ = false}};
```

Accept only msgs tagged with Ref, coming from 'lim'; buffer everything else.

The continuations are still necessary, but our sub-states are now insensitive to timing variations.

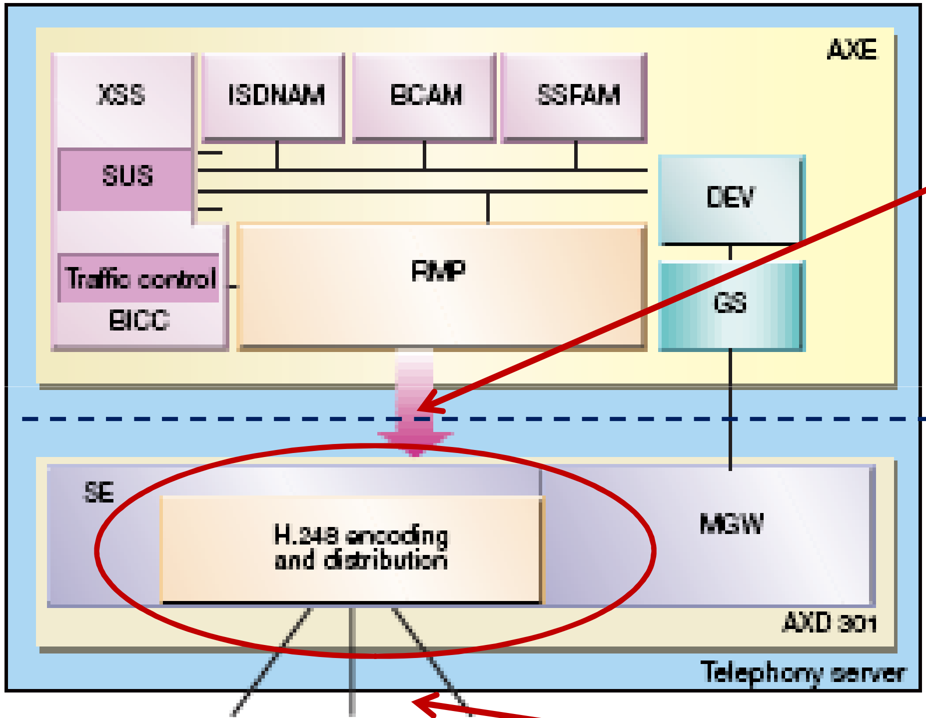
Non-blocking, using message filter (2/2)

```
event_loop(M, S, Recv) ->
   receive
      {From, Event} when element(From, Recv) == [] ->
          dispatch(From, Event, M, S);
      {From, Ref, Event} when element(From, Recv) == Ref ->
          dispatch(From, Event, M, S);
      {From, Ref, Event} when element(From, Recv) == [] ->
          dispatch(From, Event, M, S)
   end.

dispatch(From, Event, M, S) when atom(Event) ->
    handle(M:Event(From, S), M);
dispatch(From, {Event, Arg}, M, S) ->
    handle(M:Event(From, Arg, S), M).

handle({ok, NewState}      , M) -> event_loop(M, NewState);
handle({ok, NewState, Recv}, M) -> event_loop(M, NewState, Recv).
```

Properties of filtered event loop

- Can be implemented in basically any language (e.g. extending existing C++ framework.)
- Solves the complexity explosion problem.
- Doesn't eliminate the need for continuations (this affects readability – not complexity)

A (much) larger example



Short messages, chatty protocol

Legacy Phone Switch

Switch Emulator and
Voice-over-ATM Controller
("Mediation logic")

Erlang

Fewer, larger text messages

A (much) larger Example

Code extract from the AXD301-based "Mediation Logic" (ML, before rewrite)

```
%% We are waiting to send a StopTone while processing a StartTone and now
%% we get a ReleasePath. Reset tone type to off and override StopTone
%% with ReleasePath since this will both clear the tone and remove connection.
cm_msg([?CM_RELEASE_PATH,TransId,[SessionId|Data]] = NewMsg,
       HcId, #mlgCmConnTable{
                 sessionId = SessionId,
                 sendMsg = ?CM_START_TONE_RES,
                 newMsg = {cm_msg,
                           [?CM_STOP_TONE|Msg]}} = HcRec,
       TraceLog) ->
    NewHcRec = HcRec#mlgCmConnTable{
                     newMsg = {cm_msg, NewMsg},
                              toneType = off},
    NewLog = ?NewLog({cm_rp, 10}, {pend, pend}, undefined),
    mlgCmHccLib:end_session(
        pending, NewHcRec, [NewLog | TraceLog], override);
```

A (much) larger Example

Code extract from the AXD301-based ”Mediation Logic” (ML, before rewrite)

```
%% If we are pending a Notify Released event for a Switch Device, override
%% with ReleasePath.
cm_msg([?CM_RELEASE_PATH,TransId,[SessionId|Data]] = NewMsg,
       HcId,
       #mlgCmConnTable{
           sessionId = SessionId,
           newMsg = {gcp_msg, [notify, GcpData]},
           deviceType = switchDevice,
           path2Info = undefined} = HcRec,
       TraceLog) ->
   NewHcRec = HcRec#mlgCmConnTable{newMsg= {cm_msg, NewMsg}},
   NewLog = ?NewLog({cm_rp, 20}, {pend, pend}, undefined),
   mlgCmHccLib:end_session(
        pending, NewHcRec, [NewLog | TraceLog], override);
```

A (much) larger Example

Code extract from the AXD301-based "Mediation Logic" (ML, before rewrite)

```
%% Getting a ReleasePath when pending a Notify Released event is a bit
%% complicated. We need to check for which path the ReleasePath is for and
%% for which path the notify is for. If they are for different paths we are
%% in a dilemma since we only can be in pending for one of them. As a simple
%% way out we just treat this as an abnormal release for now.
cm_msg([?CM_RELEASE_PATH,TransId,[SessionId|Data]] = NewMsg,
       HcId,
       #mlgCmConnTable{
           sessionId = SessionId,
           newMsg = {gcp_msg, [notify, GcpData]},
           deviceType = switchDevice} = HcRec,
       TraceLog) ->
   mlgCmHcc:send_cm_msg(?CM_RELEASE_PATH_RES,
                        ?MSG_SUCCESSFUL, TransId, SessionId),
   NewHcRec = HcRec#mlgCmConnTable{newMsg = abnormal_rel},
   NewLog = ?NewLog({cm_rp, 30}, {pend, pend}, undefined),
   mlgCmHccLib:end_session(pending, NewHcRec,
                           [NewLog | TraceLog], override);
```

Observations

- Practically impossible to understand the code without the comments
- Lots of checking for each message to determine exact context (basically, a user-level call stack.)
- A nightmare to test and reason about
- This code has now been re-written and greatly simplified.

ML State-Event Matrix (1/4)

Triggers \ State	Null	Setup	Add	Connected	Release	ModifyCom	Modify	ToneActive	CotActive	Override	Pending	Seized	Move	Prepare	Broken
EstablishPath	1,2	y	y	40, 41, 42, 43, 44	84	y	y	123, 124	129, 130	y	y	213	220, y	y	235, 236, 237, 259
ModifyPath	y	y	y	45, 46	y	y	y	y	y	y	y	y	y	y	y
ReleasePath	4	13	y	47, 48, 49, 50, 51, 52	85, 86	13	13	y	131	134, 135, 136	150, 151, 152	y	13	13	238
StartTone	5	y	y	53	87	y	y	125	y	y	y	y	y	y	239
StopTone	5	y	y	54	88	y	111, 112, 113	126	y	137	y	y	y	y	240
PreparePath	254	y	y	55	y	y	y	y	y	y	y	y	y	y	y
BreakPath	y	y	y	56	y	y	y	y	y	y	y	y	y	y	y
SeizeDevice	6	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
ReleaseDevice	7	13	13	57, 58	89	13	13	NA	NA	138	153, 154, 155, 156	214	13	NA	241

Action procedures:

- N/A Not applicable
- x No action, ignore the error
- y Return protocol error, remain in same state
- A Anomaly, log

Alternative execution paths depending on context

ML State-Event Matrix (2/4)

State / Triggers	Null	Setup	Add	Connected	Release	ModifyConn	Modify	ToneActive	CotActive	Override	Pending	Seized	Move	Prepare	Broken
AddReply	A	A	29, 30, 31, 32, 33, 256	A	A	A	A	A	A	A	157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167	A	221	A	A
SubtractReply	A	A	A	A	90, 91	A	A	A	A	139	168, 169, 170, 171	A	A	A	A
ModifyReply	A	A	A	A	A	105, 106	114, 115, 116, 117, 118, 119	A	A	A	172, 173, 174, 175, 176, 177, 178, 179	A	A	A	A
MoveReply	A	A	A	A	A	A	A	A	A	140, 141	180	A	222, 223, 224, 225, 226	A	A
Notify - establish	x	14	34, 35, 36	59, 60	92, 93	A	A	A	A	A	181	215	227, 228	A	260
Notify - release	x	15	15	61, 62, 63, 64, 65, 66	94, 95	15	15	A	A	142	182, 183, 184, 185, 186	216	15	15	242, 243

ML State-Event Matrix (3/4)

State / Triggers	Null	Setup	Add	Connected	Release	ModifyConn	Modify	ToneActive	CotActive	Override	Pending	Seized	Move	Prepare	Broken
hc_msg - setup	8	A	A	67, 68, 69, 70, 71	96	A	A	127	132	A	187	217, 218	229, 230	A	244, 245, 246, 247, 248, 249, 250, 261, 262
hc_msg - setup_res	A	16, 17, 18, 19, 20, 21, 22, 255	A	A	A	A	A	A	A	A	188, 189, 190, 191, 192, 193, 194, 195	A	A	A	A
hc_msg - modify	A	A	A	72	A	A	A	A	A	A	196	A	A	A	A
hc_msg - modify_res	A	A	A	A	A	107, 108	120	A	A	A	A	A	A	A	A
hc_msg - release	9, 10	23	23	73, 74, 75, 76, 77	97, 98	23	23	A	A	143	197, 198, 199, 200, 201, 202, 203	A	23	23	251
hc_msg - release_res	x	A	A	A	99, 100	NA	NA	NA	NA	144	A	A	A	A	A
hc_msg - prepare	11	A	A	257	101	A	A	A	A	A	204, 205, 206, 207, 208	A	A	A	A
hc_msg - prepare_res	A	A	A	A	A	A	A	A	A	A	A	A	A	232, 233, 258	A
hc_msg - break	A	A	A	78	A	A	A	A	A	A	A	A	A	A	A
hc_msg - break_res	A	A	A	79	A	A	A	A	A	A	A	A	A	A	A

ML State-Event Matrix (4/4)

State / Triggers	Null	Setup	Add	Connected	Release	ModifyConn	Modify	ToneActive	CotActive	Override	Pending	Seized	Move	Prepare	Broken
hc_timeout	NA	24, 25, 26, 27	NA	NA	102	109	121	NA	NA	145	209	NA	NA	234	NA
gcp_timeout	A	A	37, 38, 39	A	103	110	122	A	A	146, 147	210	A	231	A	A
abnormal_rel	x	28	28	80, 81, 82, 83	104	28	28	128	133	148, 149	211, 212	219	28	28	252, 253

Observations...

Summary

- There *is* no global ordering
- Tying yourself to the actual ordering of events, leads to accidental complexity
- Complexity grows relative to the number of possible permutations of event sequences
- ...unless you have a strategy for “reordering events”
- Hard-real-time programmers basically have no choice
  - Do you?

http://github.com/uwiger/pots