Cliff Moon

Bottleneck Whack-A-Mole

Whack-A-Mole

Production Experience

Your Mileage May Vary.
This is all folklore.
Unless otherwise specified - R14B04.
Down in the weeds.

Collectors

- Terminates SSL and authenticates clients.
- Transforms IPFIX into internal formats.
- Exposes pubsub interfaces internally per customer.
- Talks to Scala nodes via Scalang (Erlang distribution protocol).

Collectors

- 2.5 years in production.
- Early versions fell over around ~700 connections, ~10k recs/sec.
- Currently handles 3000 connections, ~300k recs/sec per machine.
- ~ 100mbps ingress per machine from customers.
- ~ 300mbps egress per machine to internal network.

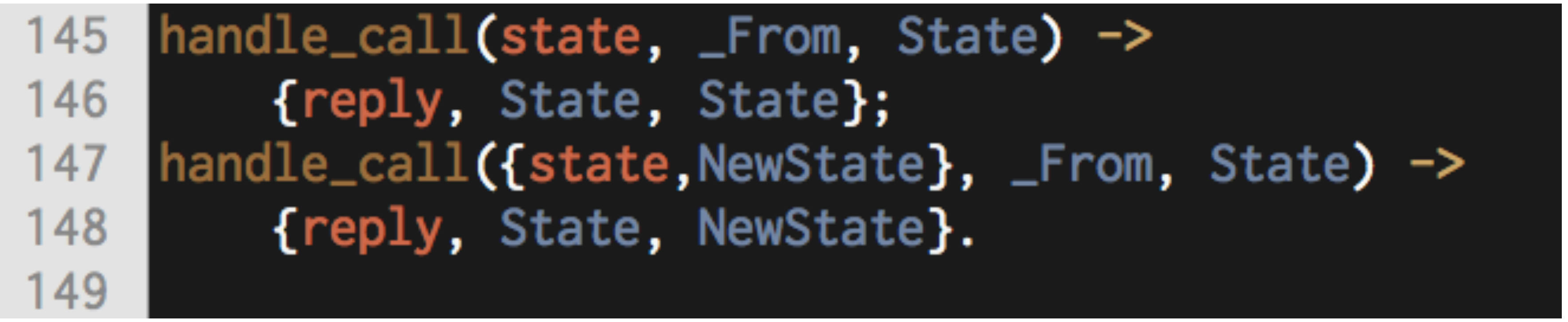
Erlang -

There is always a queue somewhere getting backed up.

Tools of the Trade

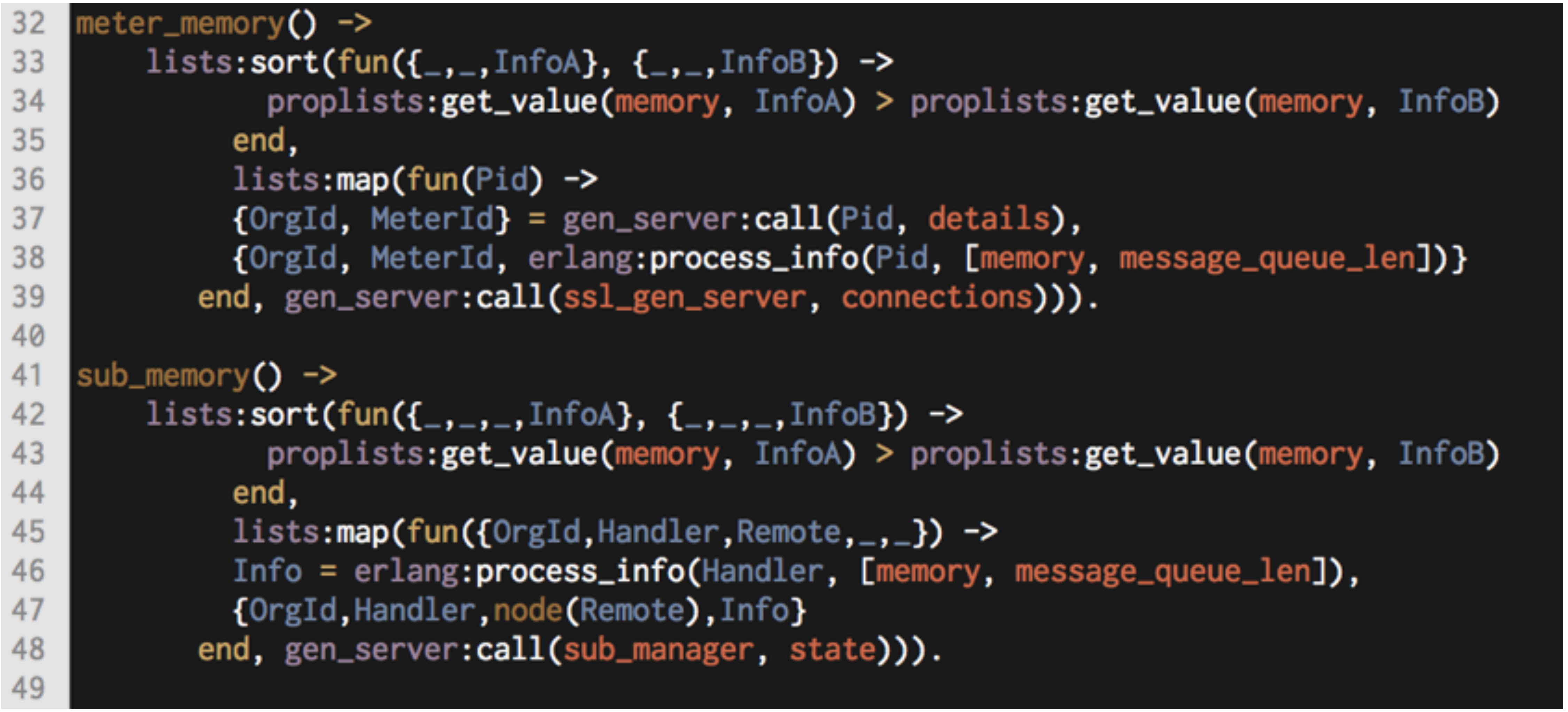
Remsh is Magical

- i().
- etop.
- process_info(pid(0,128,0)).
- process_info(Pid, [backtrace]).
- Make your own escape hatches.
- Admin functions and ops playbook for bad actors.



```erlang
handle_call(state, _From, State) ->
    {reply, State, State};
handle_call({state,NewState}, _From, State) ->
    {reply, State, NewState}.
```

Escape Hatches



```
meter_memory() ->
    lists:sort(fun({_,_,InfoA}, {_,_,InfoB}) ->
           proplists:get_value(memory, InfoA) > proplists:get_value(memory, InfoB)
         end,
         lists:map(fun(Pid) ->
         {OrgId, MeterId} = gen_server:call(Pid, details),
         {OrgId, MeterId, erlang:process_info(Pid, [memory, message_queue_len])}
       end, gen_server:call(ssl_gen_server, connections))).

sub_memory() ->
    lists:sort(fun({_,_,_,InfoA}, {_,_,_,InfoB}) ->
           proplists:get_value(memory, InfoA) > proplists:get_value(memory, InfoB)
         end,
         lists:map(fun({OrgId,Handler,Remote,_,_}) ->
         Info = erlang:process_info(Handler, [memory, message_queue_len]),
         {OrgId,Handler,node(Remote),Info}
       end, gen_server:call(sub_manager, state))).
```

Dump Memory Usage

Taxonomy of Failure

Under extreme load, what will a single process do?

Overloaded Process

For some reason an overloaded process cannot keep up with incoming message rates.

Erlang Memory Model

- Heap per process.
- Message queue is stored on the heap.
- Garbage collection puts the process to sleep.

Process Death Spiral

1. A process can do N messages / sec.
2. If the arrival rate is > N messages / sec, messages will queue.
3. Larger queues cause garbing.
4. N = N * M where M < 1.
5. Goto 1.

Garbing

"This is bad luck, the process was garbage collecting when the crash dump was written, the rest of the information for this process is limited."

Until...

```
eheap_alloc: Cannot allocate 8700015800 bytes of memory
(of type "heap").
```

Why can we not keep up?

- Receive statements.
- Doing too much work.
- Sender is too fast.

Strategies for Mitigation

boundary

Receive in gen_server

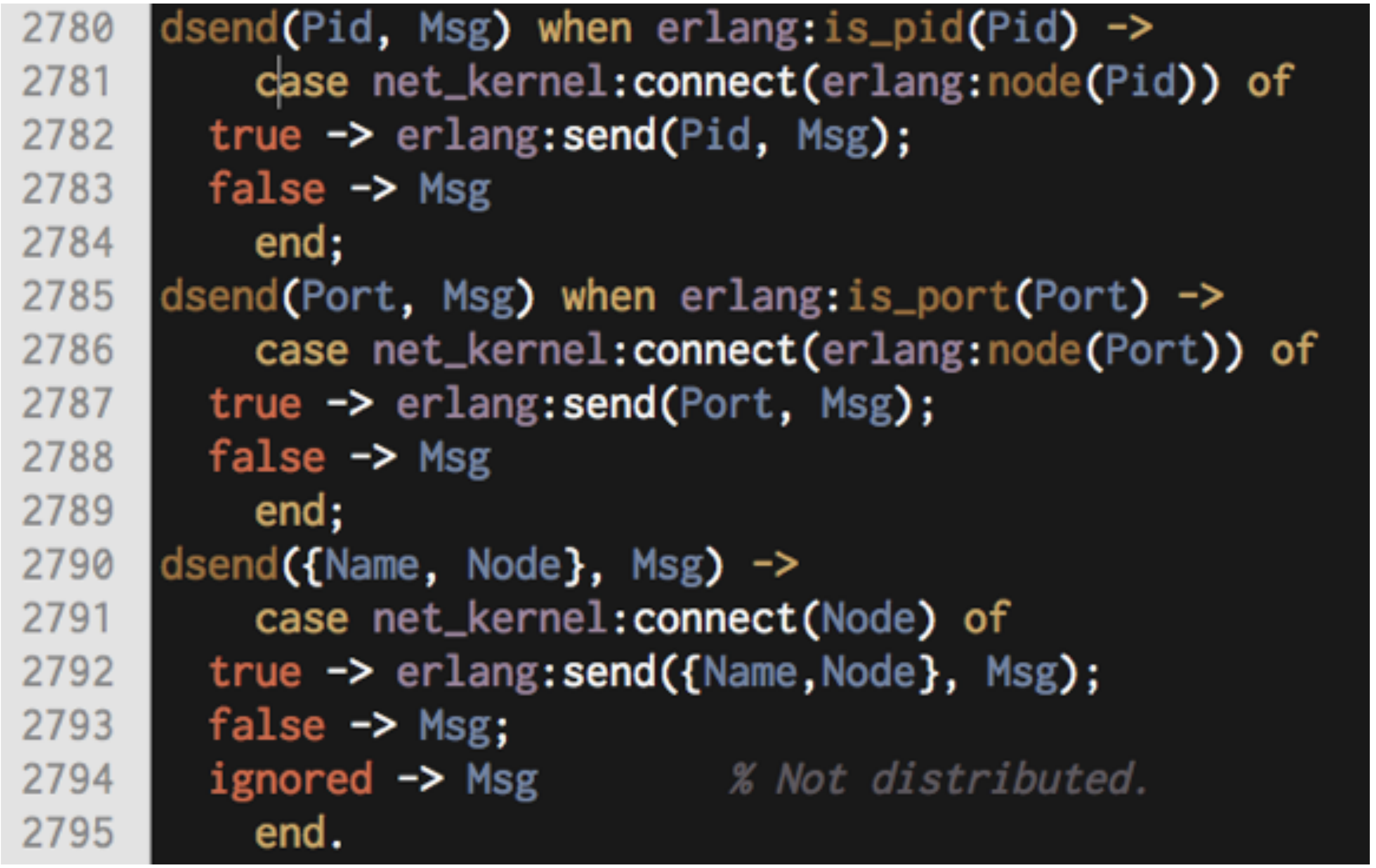
A quiz.

Which of these can cause a receive?

1. gen:call
2. gen_tcp:recv
3. Pid ! Msg

All of the above!

boundary



```erlang
dsend(Pid, Msg) when erlang:is_pid(Pid) ->
    case net_kernel:connect(erlang:node(Pid)) of
  true -> erlang:send(Pid, Msg);
  false -> Msg
    end;
dsend(Port, Msg) when erlang:is_port(Port) ->
    case net_kernel:connect(erlang:node(Port)) of
  true -> erlang:send(Port, Msg);
  false -> Msg
    end;
dsend({Name, Node}, Msg) ->
    case net_kernel:connect(Node) of
  true -> erlang:send({Name,Node}, Msg);
  false -> Msg;
  ignored -> Msg        % Not distributed.
    end.
```

What's this?

The `!` Operator!

Reducible to a gen_server:call and erlang:send.

Mitigating Errant Receives

- Separate control plane from data plane.
- Know what you are calling.
- Cut down gen_servers to as little code as possible.

Separating Control from Data

- Control needs to be low latency.
- Data needs to be high throughput.
- Separate concerns into two processes.
- Share state via ETS tables.

Doing too much.

10 pounds of sh*t in a 5 pound bag.

Do less stuff!

The preferred solution, often not feasible.

Mitigating Overload

boundary

Just Spawn a Process

- handle_call(Work, From, State) -> spawn(fun() -> gen_server:reply(do_stuff(Work), From) end),...
- Cheap GC on spawned processes.
- Can spread load across CPU’s.
- Context switching overhead.

Worker Pools

- Probably a bad idea.
- Spawning is cheap, managing a worker pool is expensive.
- Only for expensive resources like sockets, ports, etc.

Process Options

- In spawn_opt you can set min_heap_size, fullsweep_after, and priority.
- Mostly these will be fool's errands.
- Test and measure to understand the effects.

Write A NIF

- Can do work faster, can use syscalls optimized for certain workloads.
- Can also lock up the VM, segfault, abort, so forth.
- Starts a path towards C++ glued together with Erlang.
- Welp.

Fast Sender

Shut up and let me think already.

Flow Control!

Preferably explicit.

Reading from a Socket

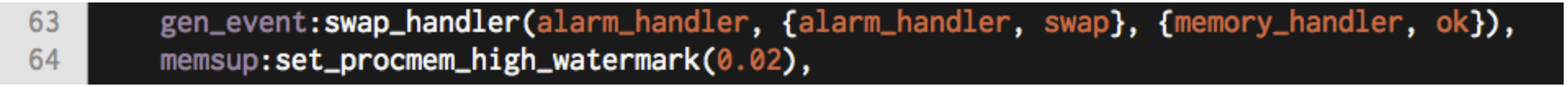
- Use {active, once}.
- Don’t use gen_tcp:recv.
- The framing socket options make this really easy.
- Buffer in the kernel TCP stack instead of your mailbox.

Process to Process

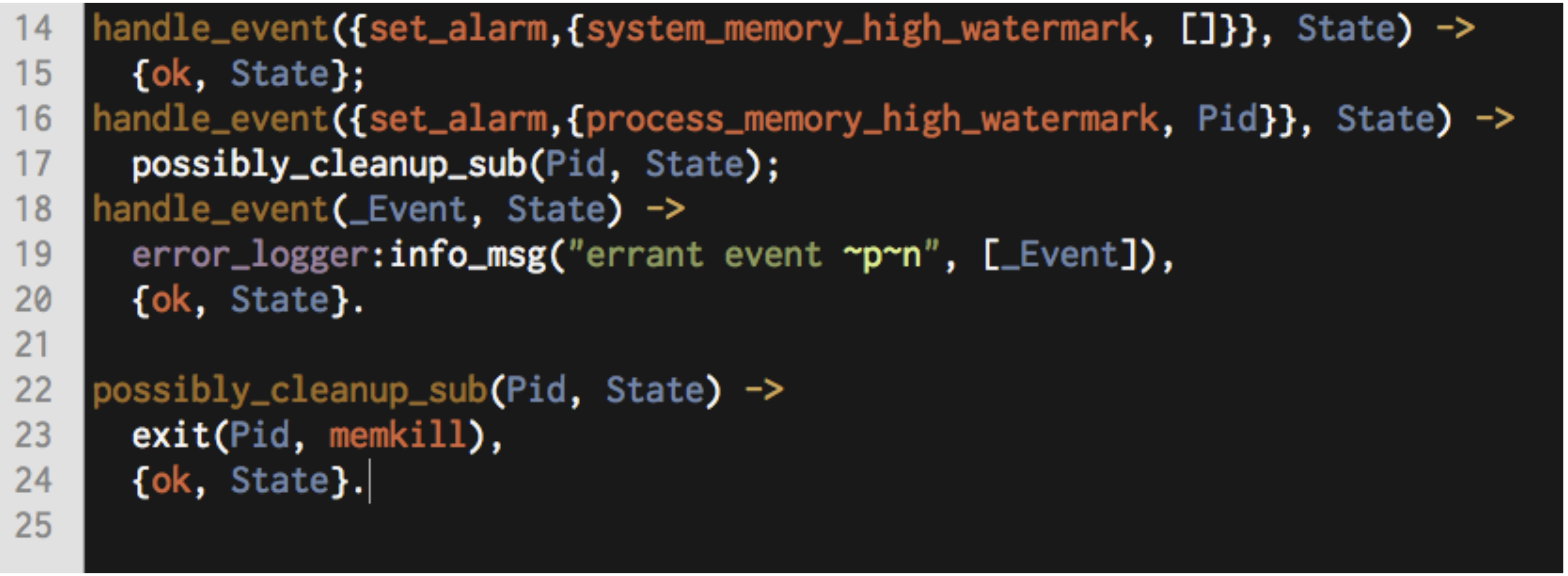
- Poor man's TCP.
- Receiver Acks every N messages.
- Sender will send N messages and wait for an ack.
- Pick a reasonable N, say 5.

Built in Flow Control

- erlang:send can sometimes suspend a process.
- When sending to a remote pid erlang:send_nosuspend might be useful.
- What's better, lose data or wait to send?



```
    gen_event:swap_handler(alarm_handler, {alarm_handler, swap}, {memory_handler, ok}),
    memsup:set_procmem_high_watermark(0.02),
```



```
handle_event({set_alarm,{system_memory_high_watermark, []}}, State) ->
  {ok, State};
handle_event({set_alarm,{process_memory_high_watermark, Pid}}, State) ->
  possibly_cleanup_sub(Pid, State);
handle_event(_Event, State) ->
  error_logger:info_msg("errant event ~p~n", [_Event]),
  {ok, State}.

possibly_cleanup_sub(Pid, State) ->
  exit(Pid, memkill),
  {ok, State}.
```

The Blowoff Valve

Memsup

- Can specify an event to fire when a process reaches a percentage of main memory.
- Execute arbitrary code in response.
- This can be used to stop the VM killing death spiral.

Mysteriously Unresponsive

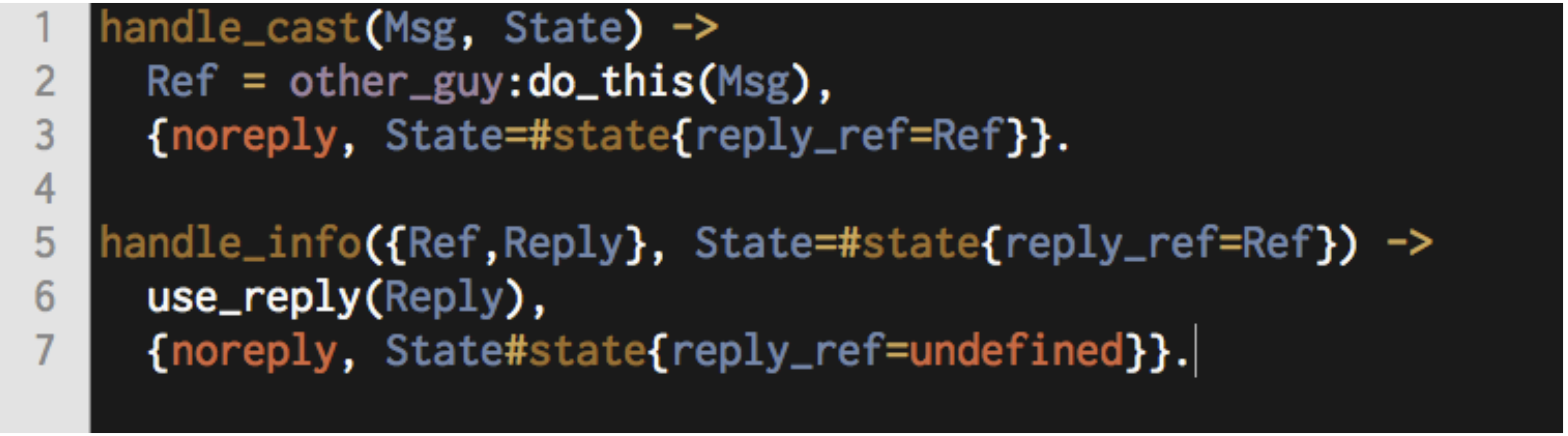
- App is not responding.
- Low resource utilization.
- What the hell is happening?

Deadlocked

- Within a given time a gen_server can process N calls.
- Your code sends it M calls where M > N.
- M-N calls will fail.
- Not dealt with, these failures will propagate.

Timeouts

- Default timeout for gen:call is 5000ms.
- Timeout of infinity can exacerbate deadlocking.
- Handle call failures.
  - Log errors.
  - Retry if appropriate.
- Does it need to be a call?



```
handle_cast(Msg, State) ->
  Ref = other_guy:do_this(Msg),
  {noreply, State=#state{reply_ref=Ref}}.

handle_info({Ref,Reply}, State=#state{reply_ref=Ref}) ->
  use_reply(Reply),
  {noreply, State#state{reply_ref=undefined}}.
```

Deferred Reply

Does it need to be a process?

Wrapping state in a process implies a mutex for accessing said state.

Refactor Processes into ETS

- Remove the gen_server and mutate an ETS table via the module API.
- Tune ETS for read concurrency or write concurrency.
- You can pass around a table reference instead of a Pid.

On to the Network

Beaten to death by runt packets.

Erlang Distribution Protocol

- Tuned for low-latency - TCP_NODELAY.
- Generally 1 message = 1 packet.
- At high throughput your network will die.

Mitigation

- Buffer in the gen_server.
  - This is a case of doing more work.
  - Can use an intermediary process.
- Just open a socket.

In Summary

boundary

Grind The Loop

- Observe that there is a problem.
- Find the overloaded queue(s).
- Mitigate the bottleneck.
- Repeat.

Questions?

Thanks.

boundary