Erlang Solutions Ltd.

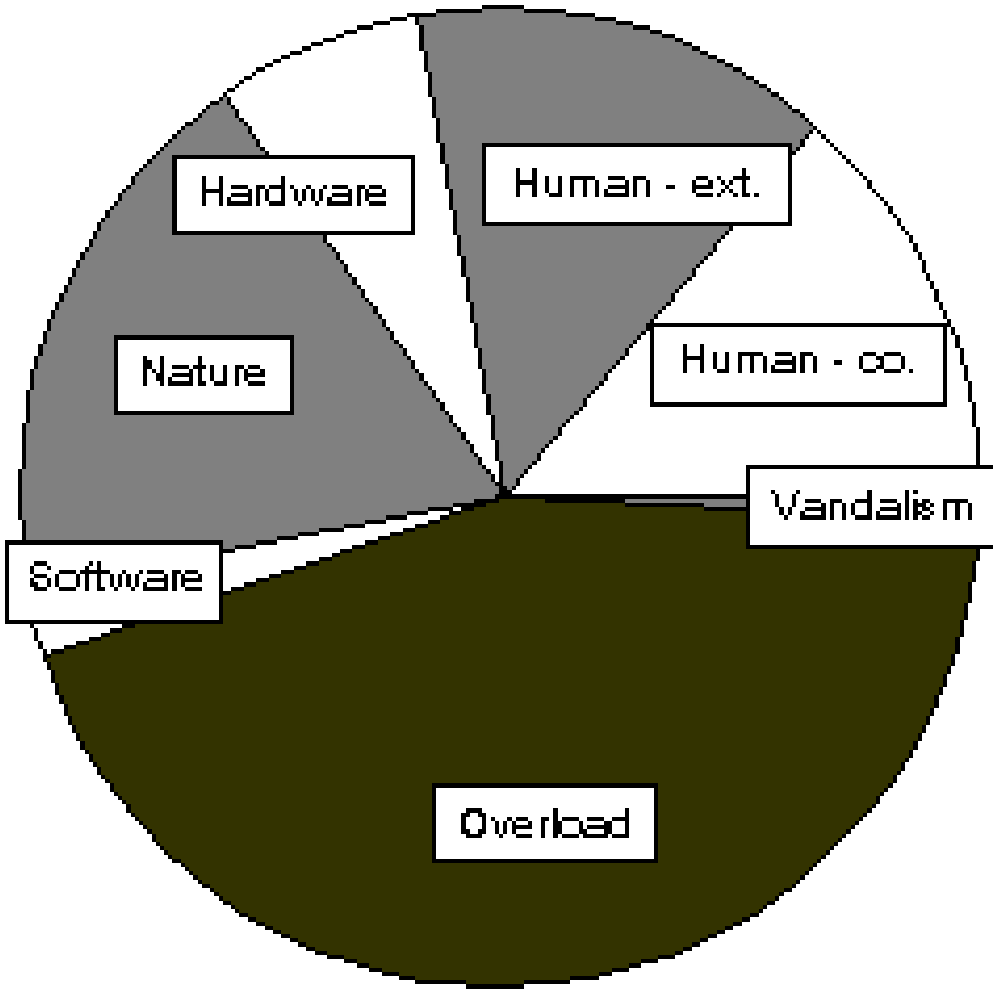
JOBS – Generic Load Regulation

Ulf Wiger
Erlang Solutions Ltd

Erlang Factory Lite, Los Angeles, 7 November 2010

Overload a Big Contributor to downtime

- In Telecoms, nearly half of registered downtime is due to overload (source: FCC)



Erlang

Erlang/OTP has no load regulation lib

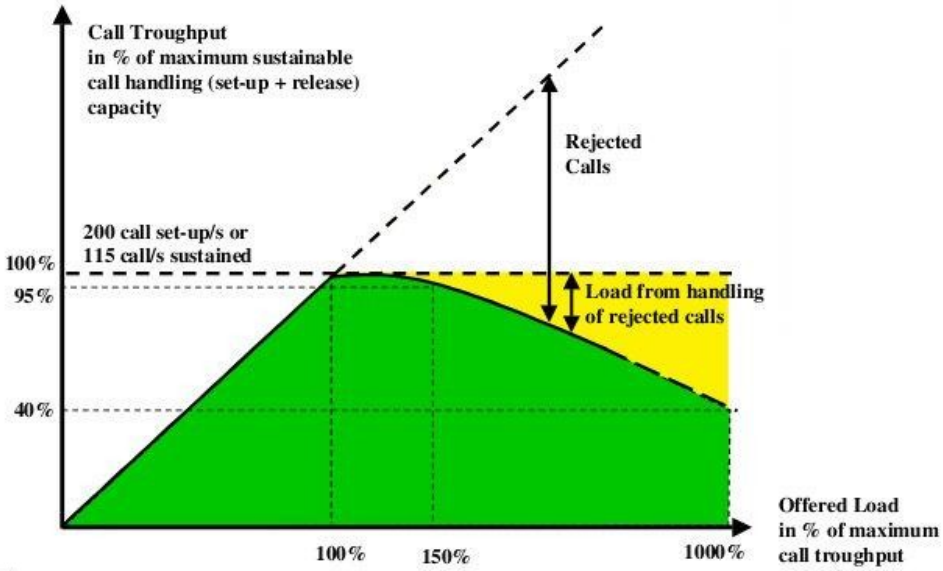
- 'overload' is an old and very primitive solution
- Estimates request frequency
- Denies request if $f > Limit$
- No queueing
- All requests are equal

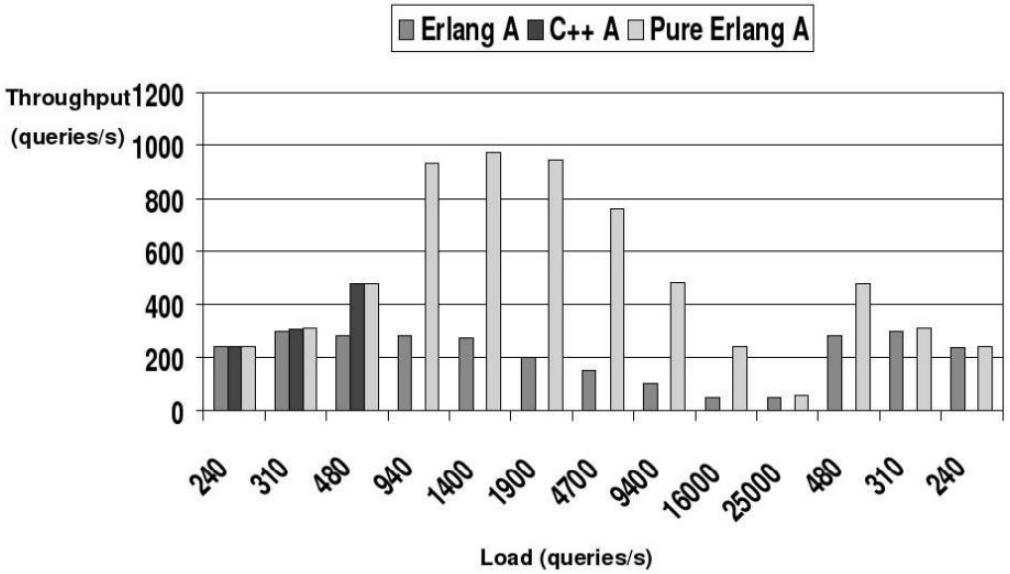
- Few know it exists – even fewer actually use it

Erlang

Erlang apps have done well in the past...

U. Wiger (2001)





J. H. Nyström, P. W. Trinder, and D. J. King (2008)

Multicore brings new exciting problems

- Emergent patterns result from non-determinism
- Risk of oscillation or memory bursts ("rouge wave" problem)
- Likely to surface during final stress tests
- Throttling (smoothing) seems to help

Photo: http://www.randyjaybraun.com

Erlang

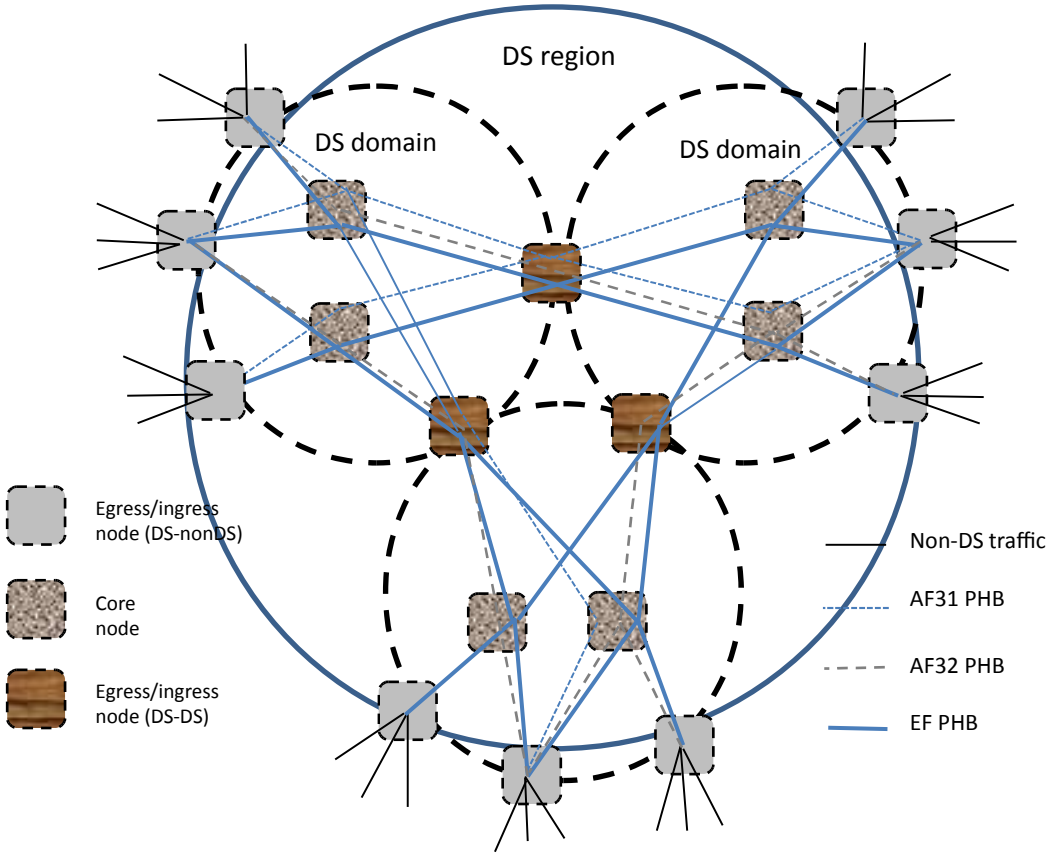
The JOBS Paper

- Common overload reasons in Erlang
- Examples of mitigation strategies
- Overview of the JOBS framework
- https://github.com/esl/jobs/raw/dev/doc/erlang07g-wiger.pdf

Erlang

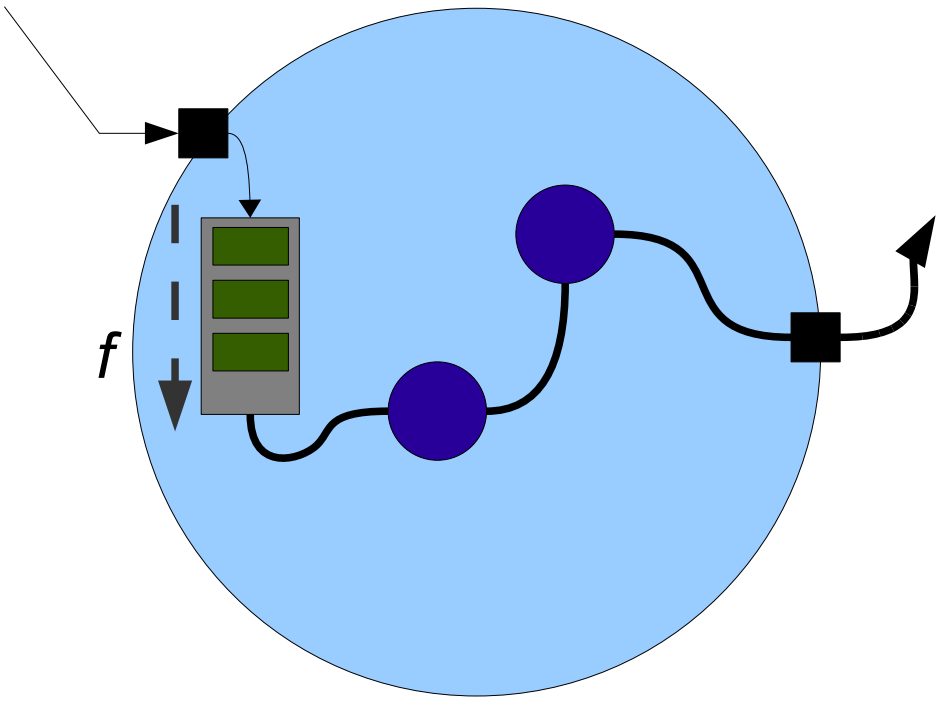
The DiffServ model from Datacom

- Stateless core
- Regulate at the edges
- Much more successful than the more complex IntServ regulation model
- Claim: This model fits well for Erlang software



Regulate at the edges

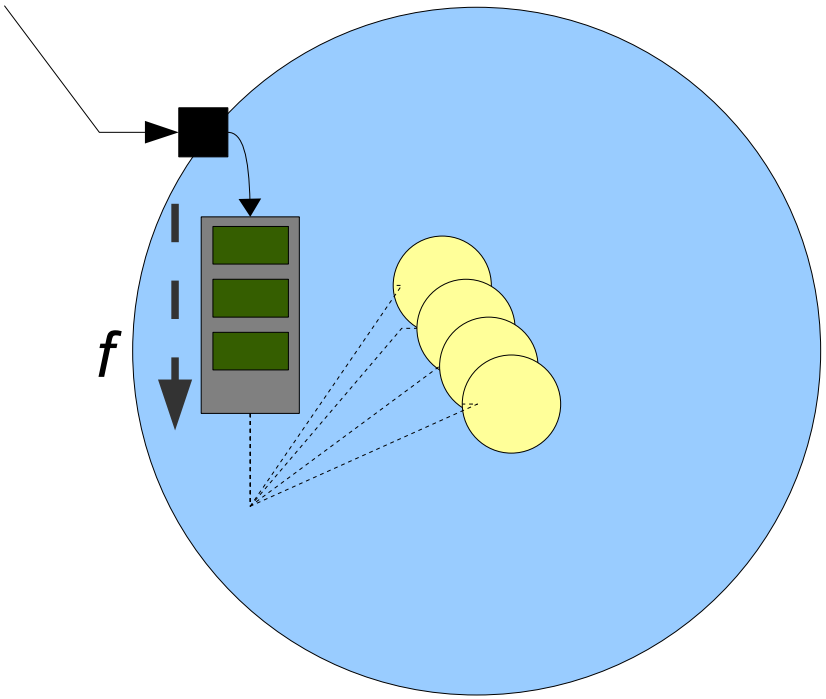
- Job input queue
  - Configurable
    - Throughput
    - Maximum wait
    - (etc...)
- “Stateless” core components



Erlang

Process pools

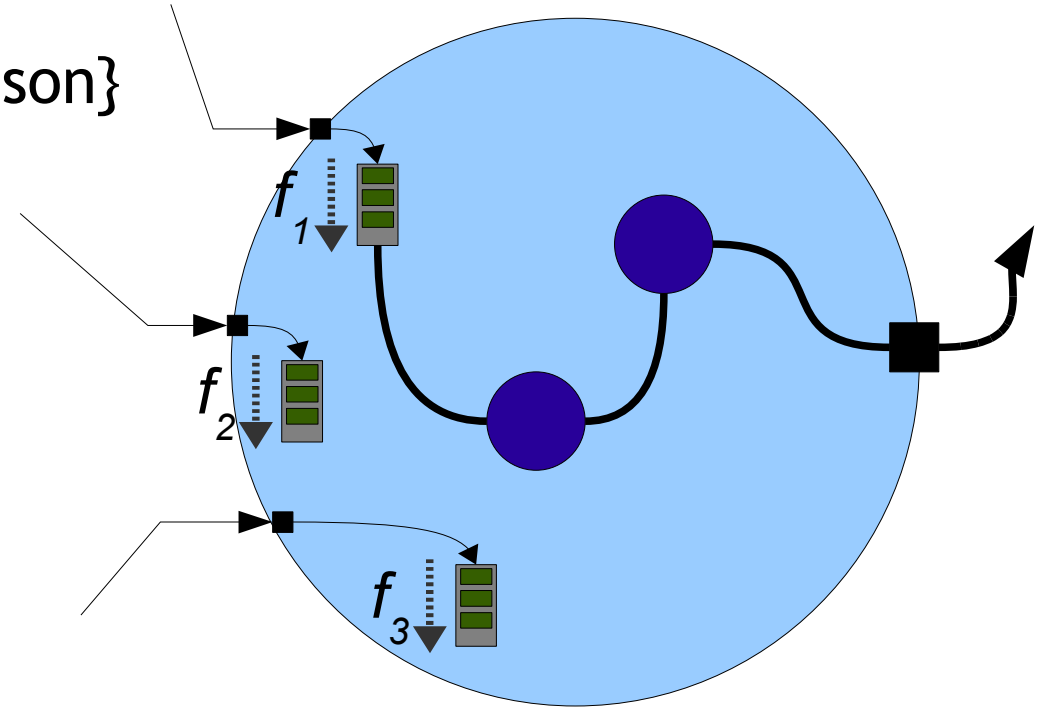
- Pool: allow N concurrent workers
  - Can actually raise throughput by lowering contention
- Credit system: check out a value from a credit pool
- 'Counter-based' regulation in JOBS combines the two
  - ~~(gproc aggregated counters)~~



Erlang

Differentiating inputs

- ask(JobType) ⟶
  {ok, Opaque} | {error, Reason}
- Set rate/lifetime per queue
- Non-rejectable jobs:
  - Infinite wait time

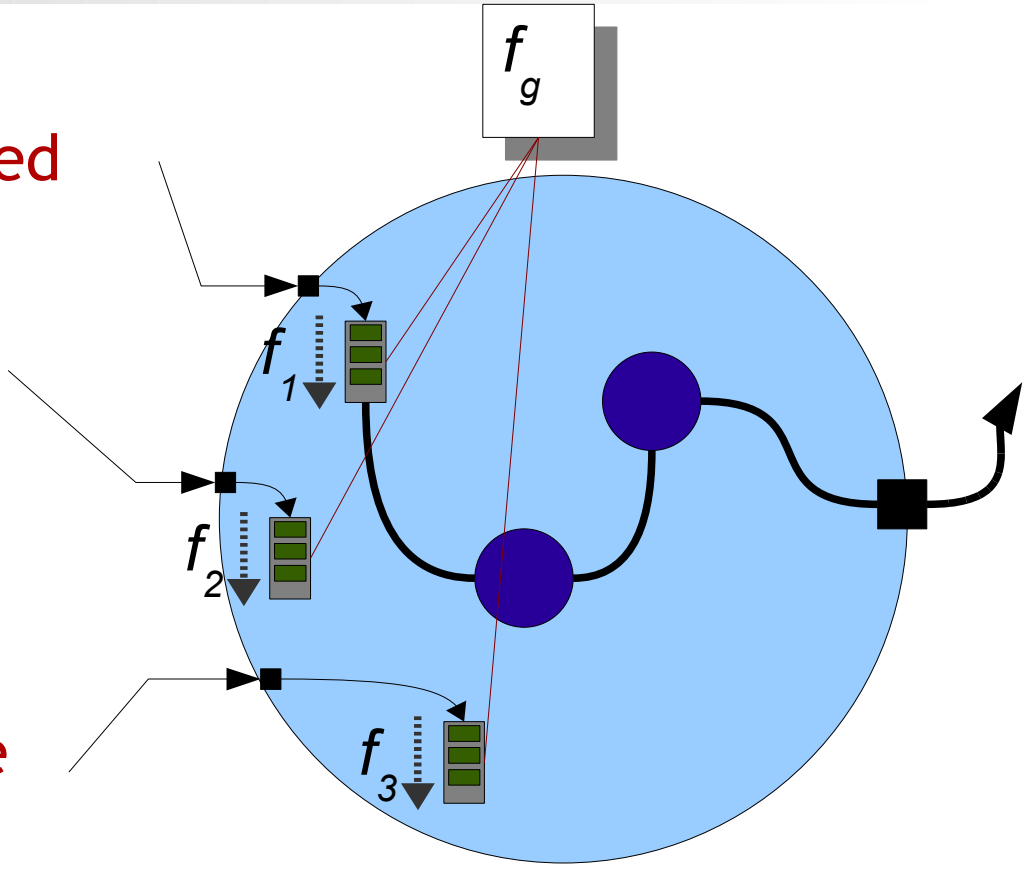


Erlang

Group rate regulation

- Total rate of jobs from grouped queues cannot exceed f_g
- Useful e.g. when serving multiple similar clients

  (or e.g. setup/release jobs)
- Counter regulators can also be shared, by giving them the same name

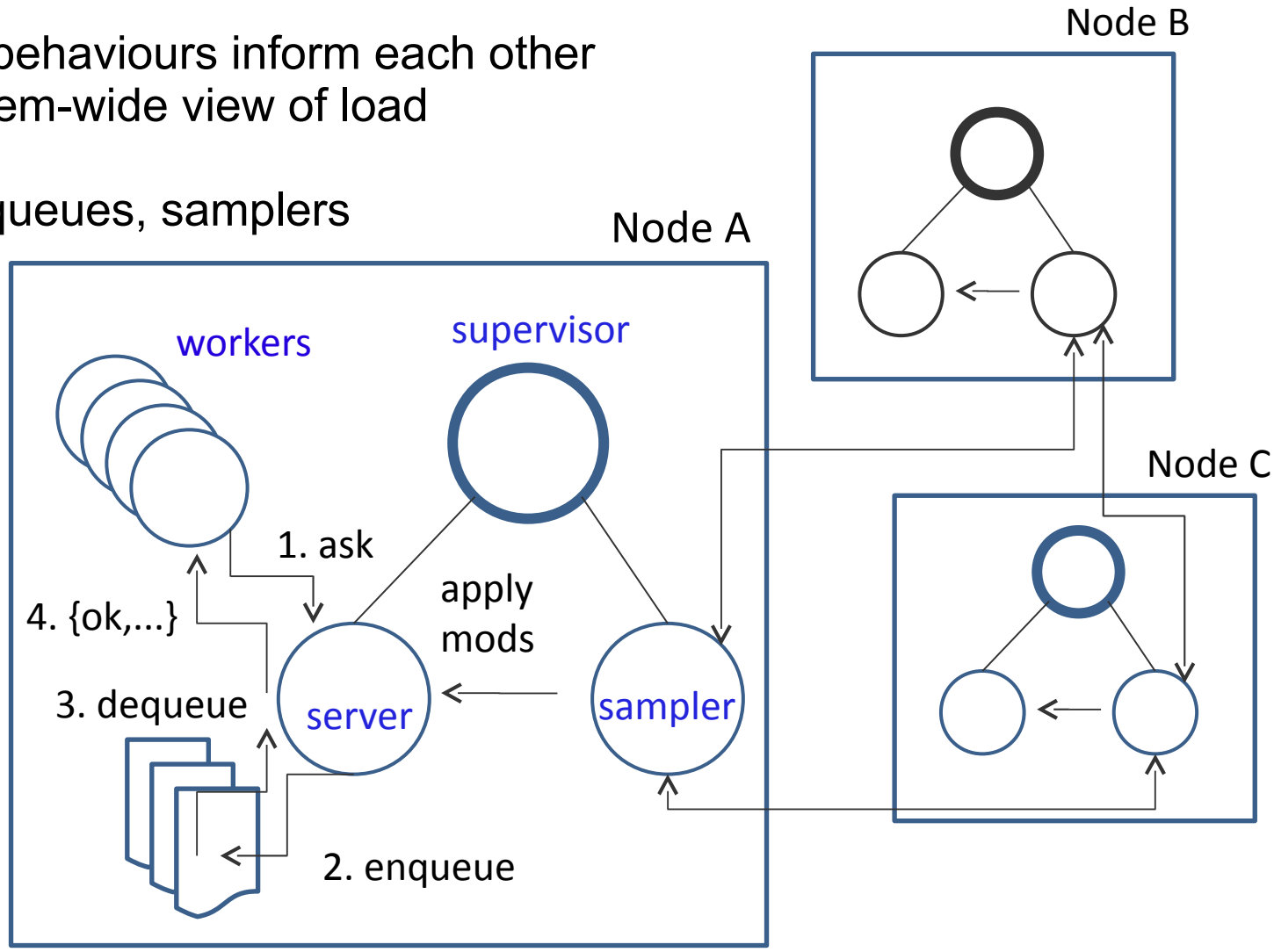


Erlang

The JOBS Architecture

Sampler behaviours inform each other for a system-wide view of load

Plugins: queues, samplers

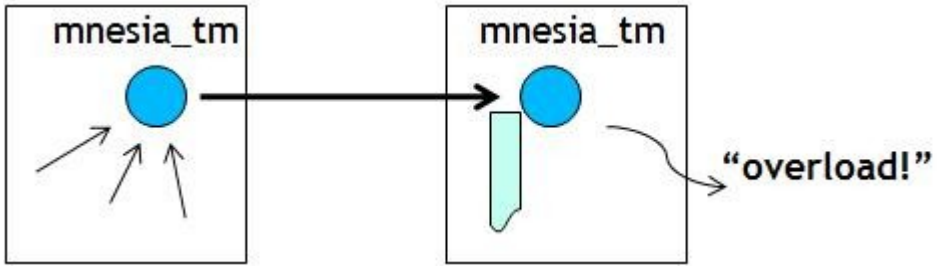


Erlang

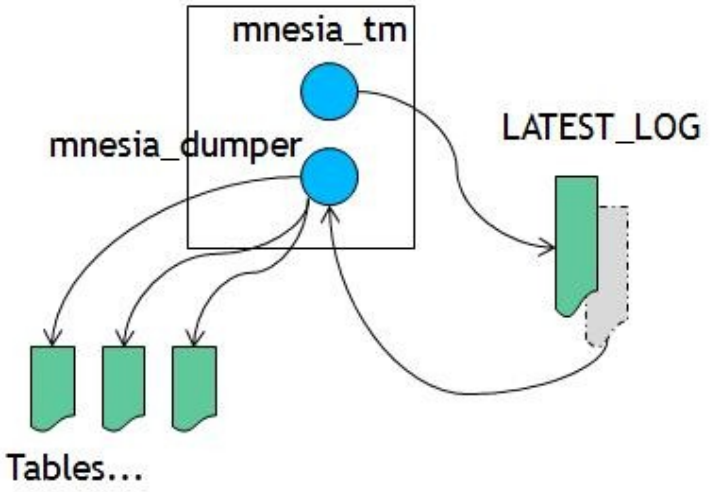
Example: Mnesia overload

- Mnesia sends 'overload' events only on the node where overload was detected
- The cause of the overload may well be on other nodes
- Feedback modifiers in JOBS lower the rate of relevant job queues

Message queue overload



Log dump overload



Erlang

Admission request function (simplified)

```
-spec ask(job_class()) → {ok, reg_obj()} | {error, rejected | timeout}.
%%
ask(Type) →
    call(?SERVER, {ask, Type, timestamp()}, infinity).


-spec done(reg_obj()) -> ok.
%%
done(Opaque) ->
    gen_server:cast(?MODULE, {done, Opaque})..
```


Erlang

Alternative request function

```
-spec run(job_class(), fun(() -> X)) -> X.
%%
run(Type, Fun) when is_function(Fun, 0) ->
    case ask(Type) of
        {ok, Opaque} ->
            try Fun()
                after
                    done(Opaque)
                end;
        {error, Reason} ->
            erlang:error(Reason)
    end.
```


Erlang

Mnesia sampler behaviour (extract)

```
init(Opts) ->
    mnesia:subscribe(system),
    Levels = proplists:get_value(levels, Opts, default_levels()),
    {ok, #st{levels = Levels}}.

default_levels() ->
    {seconds, [{0,1}, {30,2}, {45,3}, {60,4}]}.

handle_msg({mnesia_system_event, {mnesia,{dump_log,_}}}, _T, S) ->
    {log, true, S};
handle_msg({mnesia_system_event, {mnesia_tm, message_queue_len, _}}, _T, S) ->
    {log, true, S};
handle_msg(_, _T, S) ->
    {ignore, S}.

sample(_T, S) ->
    {is_overload(), S}.

calc(History, #st{levels = Levels} = S) ->
    {jobs_sampler:calc(time, Levels, History), S}.
```


Erlang

CPU sampler behaviour (extract)

```
init(Opts) ->
    cpu_sup:util([per_cpu]),  % first return value is rubbish, per the docs
    Levels = proplists:get_value(levels, Opts, default_levels()),
    {ok, #st{levels = Levels}}.

default_levels() -> [{80,1},{90,2},{100,3}].

sample(_Timestamp, #st{} = S) ->
    Result = case cpu_sup:util([per_cpu]) of
                 ...
             end,
    {Result, S}.

calc(History, #st{levels = Levels} = St) ->
    L = jobs_sampler:calc(value, Levels, History),
    {L, St}.
```


Erlang

Queue (list) behaviour (extract)

```
new(Options, Q) ->
    case proplists:get_value(type, Options, lifo) of
        lifo -> Q#q{st = []}
    end.

delete(#q{}) -> true.

in(TS, Job, #q{st = []} = Q) ->
    Q#q{st = [{TS, Job}], oldest_job = TS};
in(TS, Job, #q{st = L} = Q) ->
    Q#q{st = [{TS, Job} | L]}.

out(N, #q{st = L, oldest_job = OJ} = Q) when N >= 0 ->
    {Out, Rest} = split(N, L),
    OJ1 = case Rest of
              [] -> undefined;
              _  -> OJ
          end,
    {Out, Q#q{st = Rest, oldest_job = OJ1}}.
```


Erlang

JOBS Status

- http://github.com/esl/jobs
- Not yet fielded in real applications (coming soon...)
- More documentation, testing and debugging needed
- Feedback most welcome

Erlang