CONCURRENCY FOUNT:

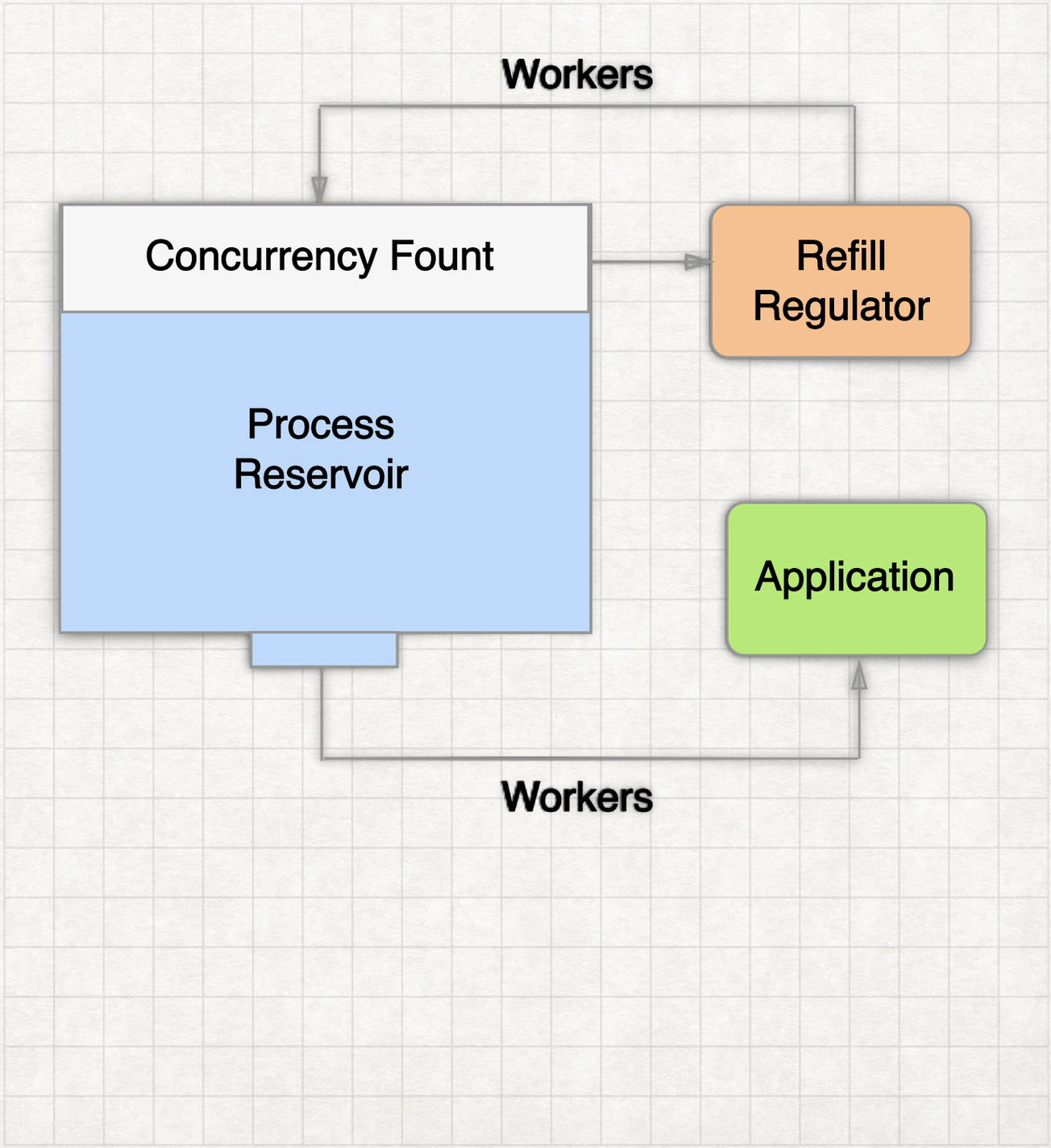
A PACE-REGULATED SUPPLY OF NEWLY SPAWNED PROCESSES

Jay Nelson https://github.com/duomark/epocxy @duomark

CONCURRENCY FOUNT

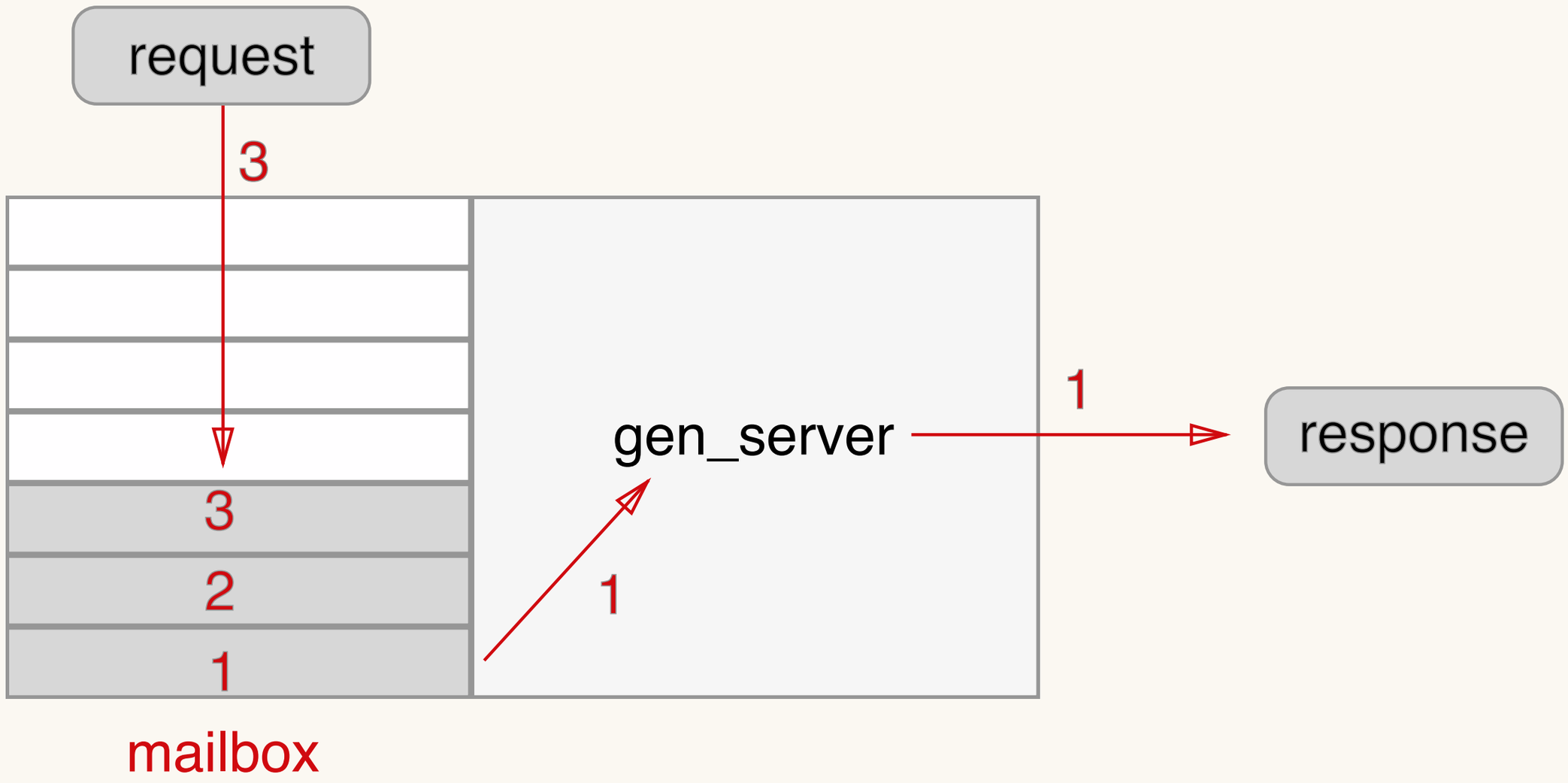
WHAT IS IT?

- A reservoir of pre-spawned processes with a regulated replacement behaviour
- A principled approach to using background tasks with back-pressure controls
- Part of Erlang Patterns of Concurrency https://github.com/duomark/epocxy



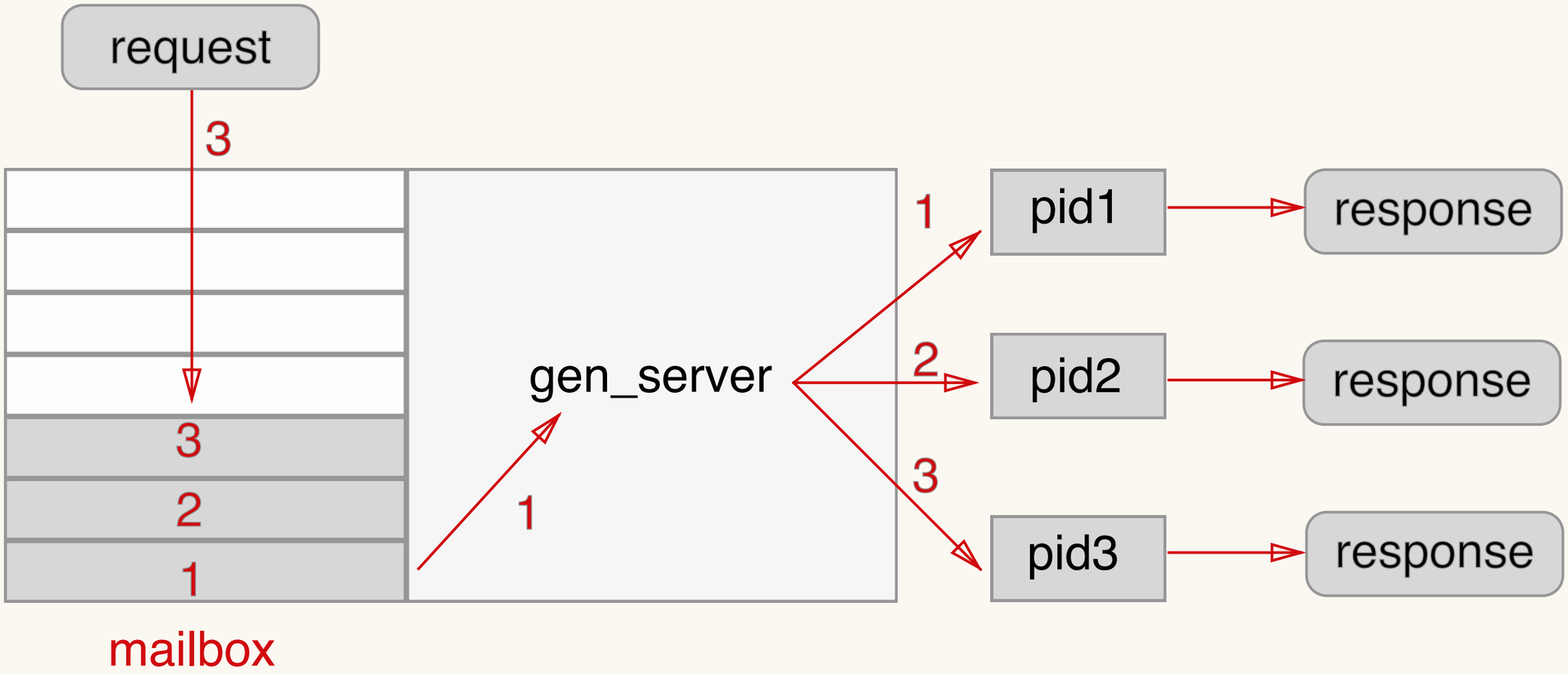
GEN_SERVER

SIMPLE CLIENT SERVER TASK



RESPONSE WORKERS

GEN_SERVER + NOREPLY TO AVOID LATENCY



RESPONSE WORKERS (CONT.)

AVOIDING CATASTROPHIC FAILURE

- What if there is a large spike in requests?
  - One worker process per request
  - Memory and CPU pressure can crash the VM
    - Cap the number of active processes
- Epocxy cxy_ctl maintains concurrency counts by category in ets
  - Refuses to execute (or runs inline) when limit exceeded
  - Still have reduced latency, without catastrophic failure risk

PERFORMANCE TUNING

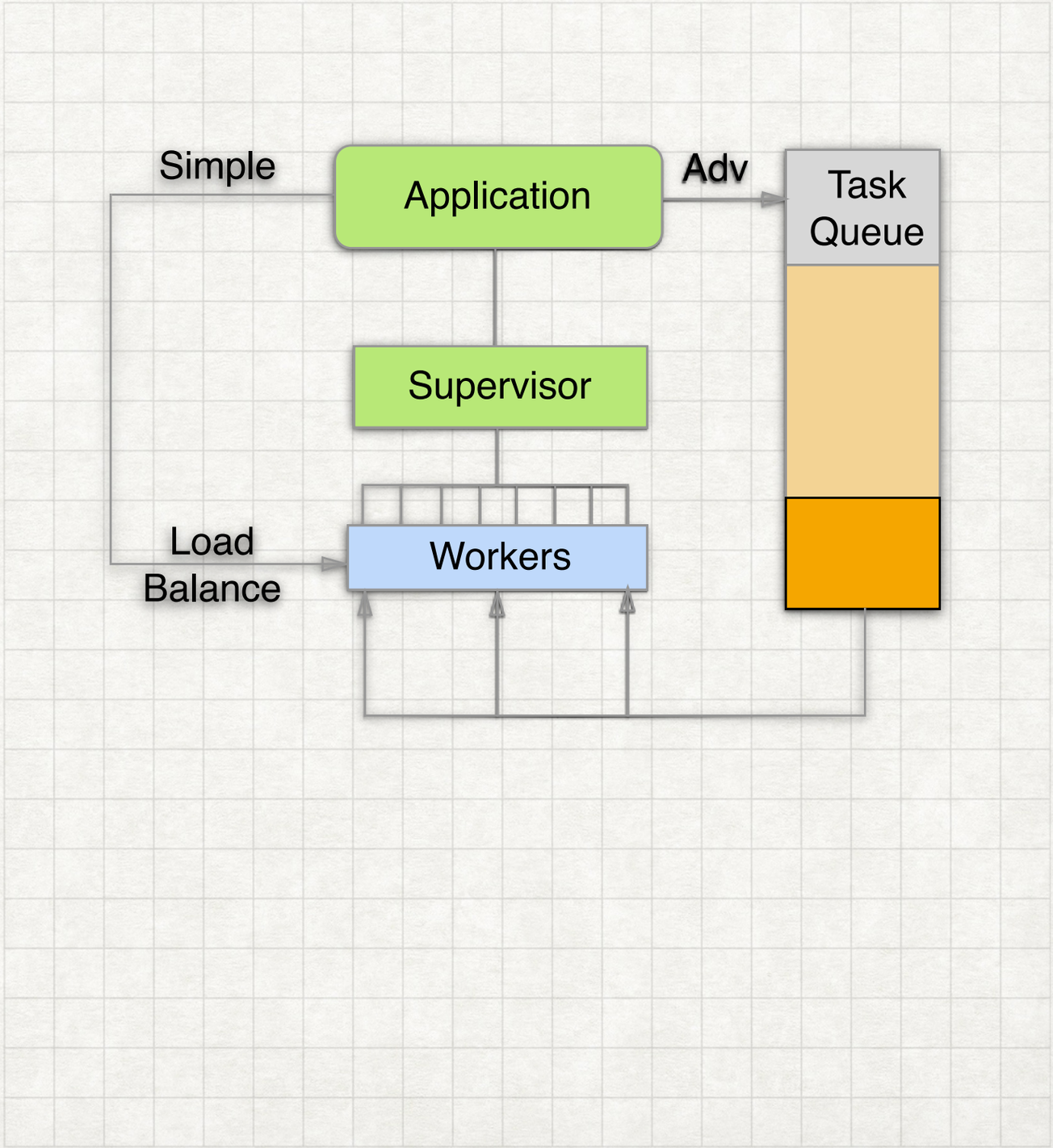
ERRATIC WORK LOAD

- Bursty traffic results in heavier CPU load
  - Cost of a spawn is concentrated; memory load heavy
  - Effects all other work done by server
- Closer look reveals short-duration tasks
  - Spawn can be 100-1000x slower than message send
  - Quick tasks benefit from pre-spawned processes
- Worker pools to the rescue!

WORKER POOLS

OVERVIEW

- Erlang makes them easy!
- Lots of libraries available!
- Until issues, then details, problems, and finally catastrophic failure…



WORKER POOL

ADVANTAGES

- Provide (typically) fixed amount of concurrency
- Workers are supervised for replacement on failure
- Tasks are usually a variety of (not dedicated) function calls
- Long-lived processes, no spawn latency
- Implemented as a library (presumably replaceable)
- Blockage on a per-worker basis if overloaded

WORKER POOL

FEATURES

- + CPU usage is predictable due to stable pool size
- + Work load spread until more tasks than workers
  - +/- Task assignment backup on mailbox common (crash loses tasks)
  - - Committed tasks can't be withdrawn from mailbox
  - - Workers may have process dictionary artifacts
  - - Process staleness issues cause fragmentation and GC
  - - No back pressure mechanism on overload
- Advanced pools: shared task queue, worker decay, forced replacement

WORKER POOL

FAILURE

- Supervisor launches replacement workers
  - Multiple failures can overload supervisor
  - Repeated failure can crash supervisor and pool itself
    - Pool crashes can trigger restart storms
    - Repeated supervisor failure takes out application
- Eliminating supervisor makes worker replacement hard
- Bottleneck issues are pervasive in naive implementations
- Supervisor generally only exists to limit pool size

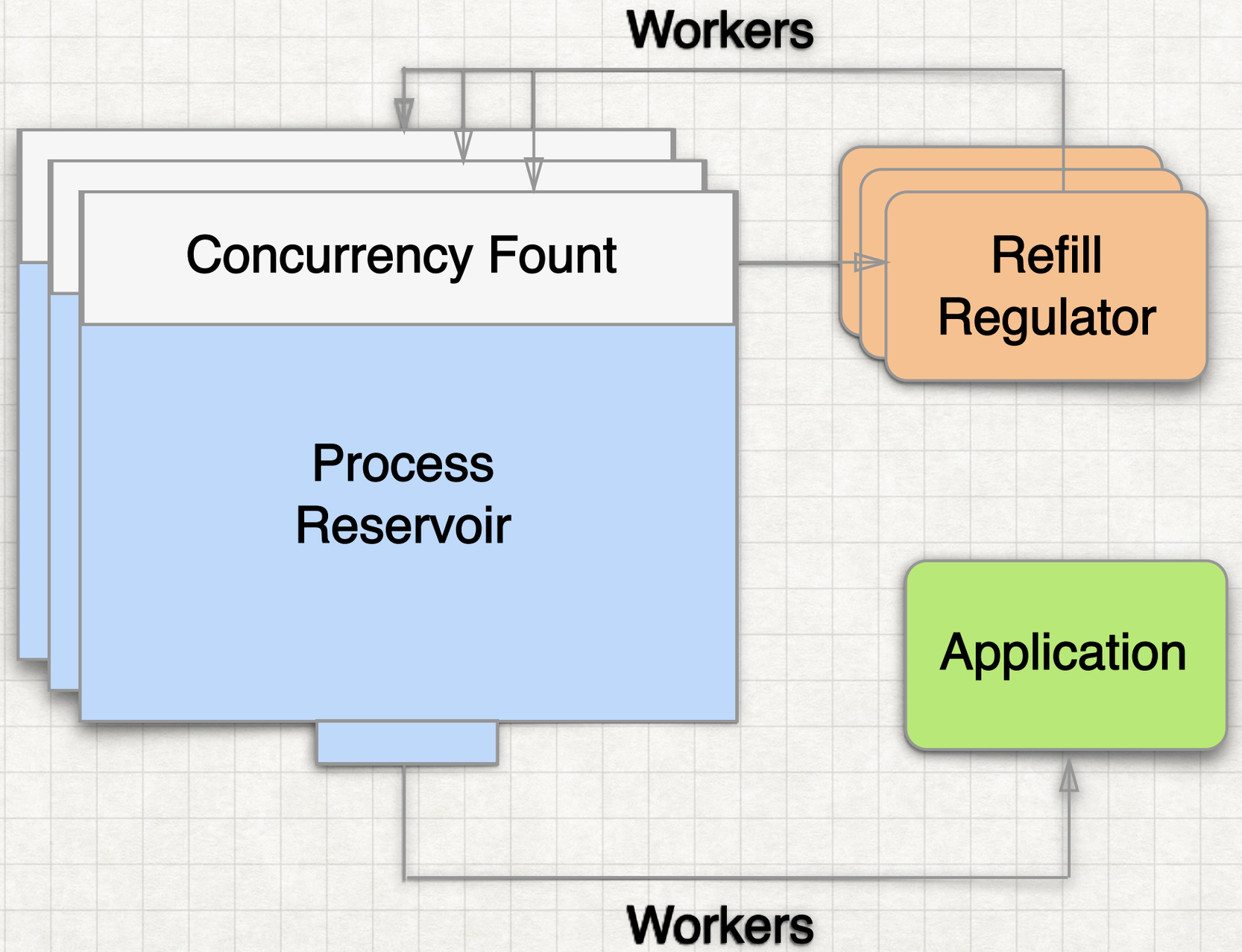
WORKER POOL

FAILURE (CONT.)

- Load balancing not easy (round-robin, random, sampled)
  - Can't predict how long worker will be busy
- Solution: Task assignment only when worker is idle
- Shared worker queue buffer of pending tasks
- Yet another coordination/locking mechanism
- Protection of queue in face of failure
- Supervisor restart has to deal with pending queue issues

CONCURRENCY FOUNT

LOAD REGULATED WORKERS



CONCURRENCY FOUNT

RESERVOIR OF PRE-SPAWNED PROCESSES

- One reservoir per task category
  - Limits request spike impact per subsystem
  - Single use process with explicitly defined behaviour
- All unallocated processes linked to reservoir
  - Allows quick cleanup of idle workers
  - No supervision needed (shouldn’t fail while idle)
  - Tasked processes have no relation to reservoir

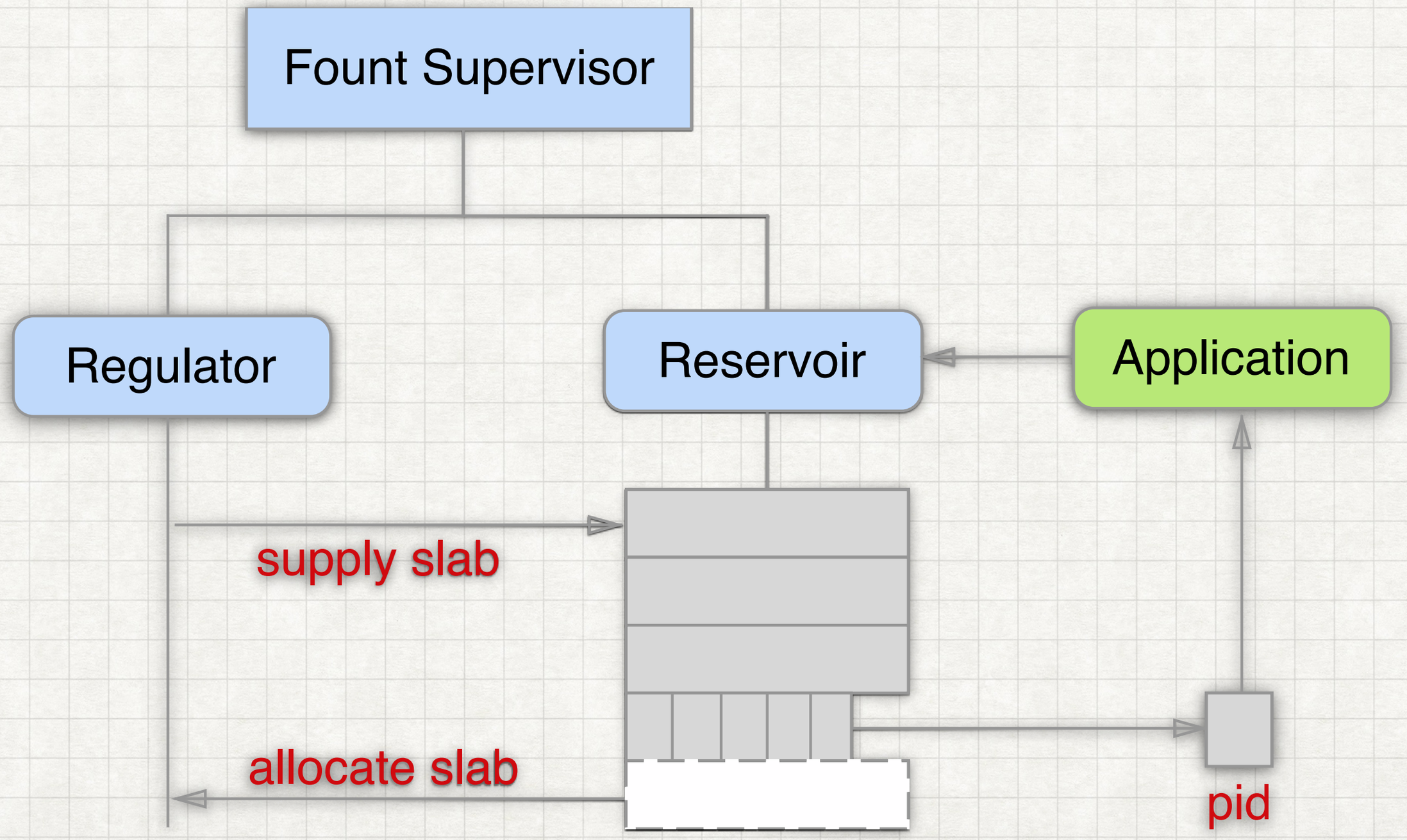
CONCURRENCY FOUNT

TASK ASSIGNMENT

- Allocation always returns pid or list of pids
  - Unlinks returned processes from reservoir
  - Optionally sends message to pid(s) before returning them
- Caller can relink to active computation processes
- Consumption triggers request for replacement processes
  - Spawn rate regulated by a separate gen_fsm
  - Currently one slab per 1/100th of a second
  - Ultimately will be customizable regulator behaviour

CONCURRENCY FOUNT

PROCESS REPLACEMENT



CONCURRENCY FOUNT

RESERVOIR IMPLEMENTATION

- Slabs of processes
  - Specify number and size of slabs on initialization
  - Dictates size of spike that can be absorbed
  - Latency occurs only when reservoir depleted
  - Task allocation efficiently manages slabs
  - Replacement only occurs as a slab-sized batch of processes

CONCURRENCY FOUNT

REGULATOR IMPLEMENTATION

- Receives replacement requests from reservoir
  - Paces the replacement at one slab per 100th of a second
  - Pace can be adjusted by tuning slab size vs depth
  - Spawning cost doesn’t impact client requests
- Messages full slabs back to the reservoir
  - Avoids flooding reservoir with messages
  - Interleaves replacement with client requests

CONCURRENCY FOUNT

FOUNT INTERFACE

- Get one or more processes -> returns [pid()] from idle reservoir
- Task one or more processes with [Msgs] -> returns [pid()] from idle reservoir
  - Each message supplied is passed to a separate idle process
- Timer, repeat, or continuation retry executed on client call
  - Provides back-pressure by pausing on retry
  - Gives up if not enough resources
  - Caller can react to [] reply
- Get status of fount
- Get timing: spawn rate, slab rate, etc.

CONCURRENCY FOUNT

FOUNT MONITORING

- Register one gen_event with the reservoir which notifies:
  - when slab of pids added (to monitor replacement pace)
  - when reservoir empty and pid needed (for overload notification)
  - when regulator reference changed (dynamic pacing)
  - when unknown messages arrive (log or debug errors)
  - when stopped
- No gen_event is registered by default

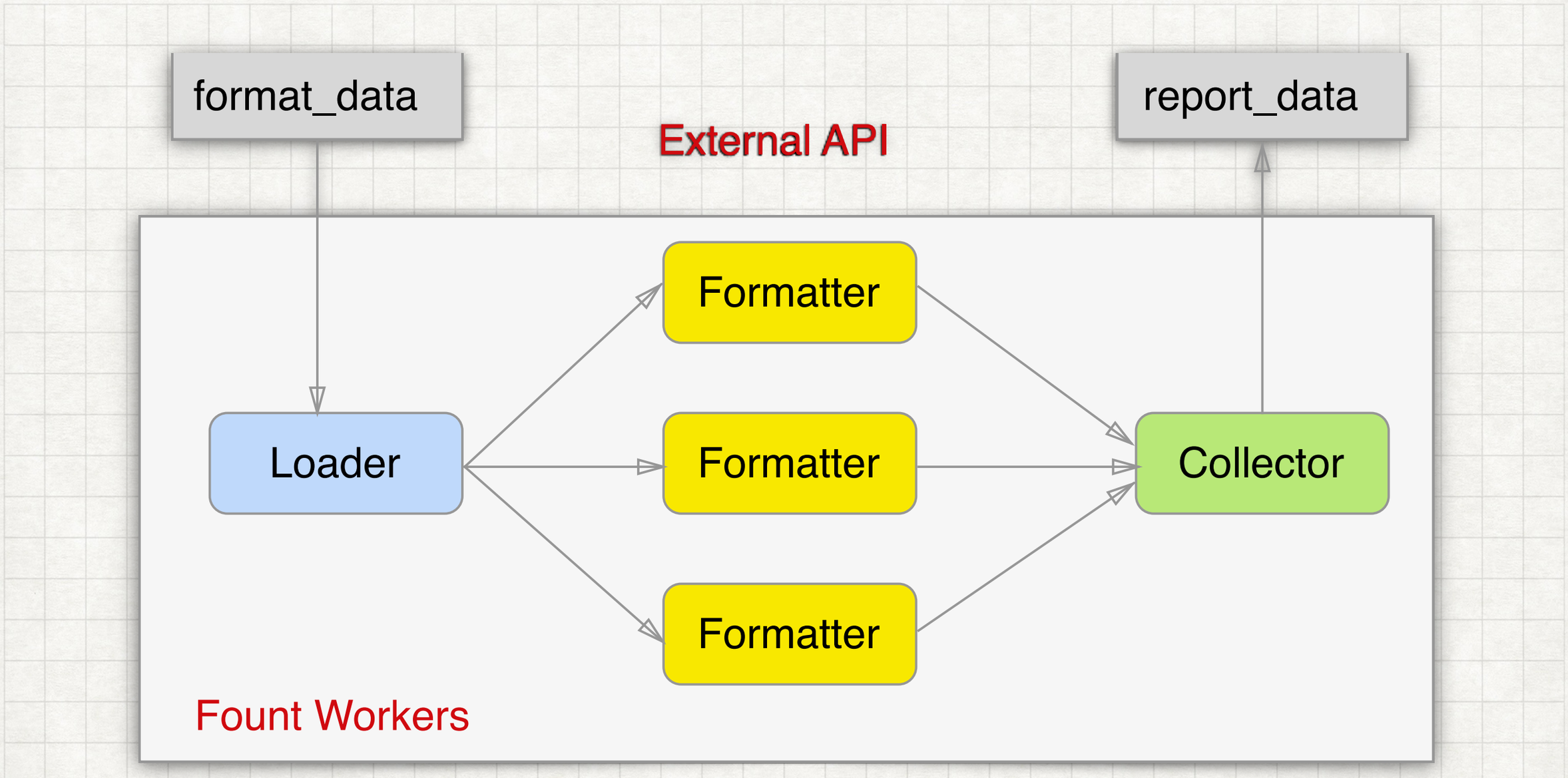
CONCURRENCY FOUNT

BENEFITS

- Each worker category requires a behaviour (can be shared across categories)
- Tasked processes are independent of idle processes
  - Tasked workers not supervised or linked to application by default
  - Idle workers can be replaced or eliminated easily (dynamic resize reservoir)
  - One-shot use avoids staleness issues with workers
  - Reservoir can absorb spikes beyond normal busy size
- Avoids overload through regulated replacement
  - Pacing enables adaptive, efficient processing
- Provides back-pressure signals for caller decision-making

EXAMPLE USING FOUNT

CONCURRENT PER LINE HEXDUMP



Loader splits data to lines, so each worker can format one line

CONCURRENT HEXDUMP

FOUNT BEHAVIORS

```
%%% create module-specific state inside idle reservoir worker
-spec init({}) -> {}.
init({}) -> {}.    % No module state needed

%%% called by regulator to create pids when replacing a slab
-spec start_pid(cxy_fount:fount_ref(), {}) -> pid().
start_pid(Fount, State) ->
    cxy_fount:spawn_worker(Fount, ?MODULE, formatter,
                           [Fount, State]).

%%% called by API when using cxy_fount:task_pid
-spec send_msg(Worker, hexdump_cmd())
          -> Worker when Worker :: worker().
send_msg(Worker, Msg) ->
    cxy_fount:send_msg(Worker, Msg).
```

CONCURRENT HEXDUMP

COLLECT AND REFORMAT RESULTS

```
%%% Send data to be reported
-spec format_data (cxy_fount:fount_ref(), binary(), pid()) -> [pid()].
format_data(Fount, Data, Hex_Collector) ->
    cxy_fount:task_pid(Fount, {load, Data, Hex_Collector}).

%%% Report results from the final format collector process
-spec report_data  (pid()) -> string() | no_results.
report_data() ->
    receive {hexdump, Lines} ->
            lists:flatten(
              [io_lib:format("  ~p.  ~s ~s  |~s|  ~p~n",
                             [Index, Address, Hexpairs, Window, Pid])
               || {Index, Address, Hexpairs, Window, Pid} <- Lines])
    after 1000 -> no_results
    end.
```

CONCURRENT HEXDUMP

WORKER RECEIVE LOOP

```
%%% a single message arrives after unlinking from fount
%%% the freed worker runs to completion
formatter(Fount, {}) ->
  %% Fan out of workers: 1 loader, N formatters, or 1 collector
  %% Each responds to one particular message only.
  receive

    %% First stage data loader
    {load, Data, Caller} when is_binary(Data), is_pid(Caller) ->
      Lines = split_lines(Data, [], 0),
      Num_Workers = length(Lines),
      [Collector | Workers]
          = cxy_fount:get_pids(Fount, Num_Workers+1),
      Collector ! {collect, Num_Workers, Caller},
      done = send_format_msgs(Workers, Lines, Collector);
```

CONCURRENT HEXDUMP

WORKER RECEIVE LOOP (CONT.)

```
%% Data formatting worker
{format, Position, Address, Line, Collector} ->
  Collector ! {collect, Position, addr(Address),
               hex(Line), Line, self()};

    %% Collector
    {collect, Num_Workers, Requester} ->
      collect_hexdump_lines(array:new(), Num_Workers, Requester)
end.
```

CONCURRENT HEXDUMP

EXAMPLE EXECUTION

```
1> {ok, P1} = cxy_fount_sup:start_link(hexdump_fount, [{}]).
{ok,<0.13657.1>}

2> cxy_fount_sup:get_fount(P1).
<0.13659.1>

3> hexdump_fount:format_data(v(2), <<"This is intended to be a fairly long line of text">>, self()).
[<0.13679.1>]

4> io:format(“~s”, [hexdump_fount:report_data()]).
  0.  00000000 54 68 69 73 20 69 73 20 69 6e 74 65 6e 64 65 64  |This is intended|  <0.13674.1>
  1.  00000010 20 74 6f 20 62 65 20 61 20 66 61 69 72 6c 79 20  | to be a fairly |  <0.13675.1>
  2.  00000020 6c 6f 6e 67 20 6c 69 6e 65 20 6f 66 20 74 65 78  |long line of tex|  <0.13676.1>
  3.  00000030 74                                                |t               |  <0.13677.1>
```

CONCURRENCY FOUNT

CONCLUSION

- Worker pools provide generic reused processes
  - Overload leads to lost tasks, or complicated queuing
  - Hierarchy and bottlenecks lead to catastrophic failures
  - Back-pressure is not intrinsic, hard to add to pool
- Concurrency fount is a stream of fresh, independent processes
  - Application is free to combine them dynamically
  - Simplicity in mechanism removes bottlenecks and hierarchy
  - Pace-regulation of spawns provides back-pressure naturally