How to Pick a Pool in Erlang without Drowning

Anthony Molinaro
OpenX, Inc.

A little about me

- I’ve been doing server side web development professionally since 1996.
- I’ve been doing high volume low latency server side web development since 1998
- Since 2008, this has been exclusively in Erlang

Assumptions about you

- You know a little Erlang
- You've written a gen_server or two
- Curious about pooling of resources (particularly process identifiers but in general any sort of state).

A few things about Erlang

- Built around Concurrency
  - Light weight processes
  - Shared nothing message passing
- A common server pattern is an Erlang process per action (most often a request)
  - State can be a bottleneck when it needs to be shared

What sort of state might you want to share?

- Configuration
- Processed/cached contents of a large file
- Large data structure
- Persistent connections to another system

What sort of options do you have for sharing state?

- Recreate every time you need it
- Use ETS/DETS/Mnesia
- Use an external cache/DB (Riak/Redis/MySQL/Memcache/etc) and make network calls
- Put it in a process

If it's in a process you can...

- Send a message to the process to get the state.
- Send a message to the process to set the state.
- Send some parameters to a process to combine with state and compute something.
- Keep a connection as the state of a process and send messages to the process to communicate over the connection.

Get access to some shared State

```
-module (stuff).

start_link () ->
  gen_server:start_link ({local, ?MODULE}, ?MODULE, [], []).

get_state () ->
  gen_server:call (?MODULE, {get_state}).

init ([]) ->
  State = get_state_from_somewhere (),
  {ok, State}.

handle_call ({get_state}, _From, State) ->
  {reply, {ok, State}, State}.
```

Do some work based on some State

```
-module (stuff).

start_link () ->
  gen_server:start_link ({local, ?MODULE}, ?MODULE, [], []).

search (Params) ->
  gen_server:call (?MODULE, {search, Params}).

init ([]) ->
  Tree = get_large_search_tree_from_somewhere (),
  {ok, Tree}.

handle_call ({search, Params}, _From, Tree) ->
  Answer = search_in_tree (Params, Tree),
  {reply, {ok, Answer}, State}.
```

Send a request across a shared connection.

```
-module (stuff).

start_link () ->
  gen_server:start_link ({local, ?MODULE}, ?MODULE, [], []).

get_data (Query) ->
  gen_server:call (?MODULE, {get_data, Query}).

init ([]) ->
  Connection = connect_to_somewhere (),
  {ok, Connection}.

handle_call ({get_data, Query}, _From, Connection) ->
  Answer = query (Query, Connection),
  {reply, Answer, Connection}.
```

Looks good, but what's the drawback?

- Concurrency
  - Process mailbox (mostly) serializes requests
    - Theoretically unlimited in length
    - Only supports basic back pressure through reduction counts
- Still mostly works, computers are fast, but does not scale across cores.

Pooling to the Rescue?

- Goto Github and search for
  - “process pool” - 12 results
  - “resource pool” - 3 results
  - “worker pool” - 18 results
  - “connection pool” - 19 results
  - Additionally, I knew about 7 more libraries not returned
- Out of these 59, only one library listed in multiple results

Whittling it down

- Does the project appear active?
  - Recent commits, recent issues
- Is it standalone/general purpose?
  - In other words it's not pool library + db connection
- Is it ready for use
  - Releases are tagged
  - Can be ingested and built by rebar without forking

Final List to explore further

- poolboy - by far the most popular
- pooler - by far the most OTP
- gen_server_pool - easy to use (and written at OpenX so I know the most about it)
- dispcount - stochastic dispatch
- gproc - pluggable dispatch models

Considered, but had a few problems

- leo_pod - interesting because it claims issues with ETS
- sidejob - interesting way to dispatch work based on scheduler locality
- pq - interesting because it uses a gen_fsm for dispatch
- episcina - looks promising, but failed to compile as a rebar dependency
- worker_pool - from Inaka, it's had some blog posts and been used in production, but doesn't tag releases

Common Components

- Worker Pool - A supervised/monitored set of processes

- Dispatching - Some strategy for selecting one of the workers

Features which can differentiate

- Ease of use
- Features of Worker Pool
    - Variable size (min/max)
    - Auto size (grow/shrink)
        - Shutdown based on age
        - Shutdown based on idle
- Features of Dispatching
    - Method (checkin/checkout/random/round robin/etc).
    - Queue or Fast Fail
- Performance?

Getting into some Details

- Example Worker

```
-module (pt_baseline_worker).

-export ([ start_link/1, do/3 ]).

start_link(WorkerArgs) ->
  gen_server:start_link(?MODULE, WorkerArgs, []).

do (Pid, N, Data) ->
  gen_server:call (Pid, {work, N, Data}).

handle_call ({work, N, Data}, _From, State) ->
  { reply, {ok, pt_util:work (N, Data)}, State}.
```

- Use fixed size and fail fast semantics.

Overview - poolboy

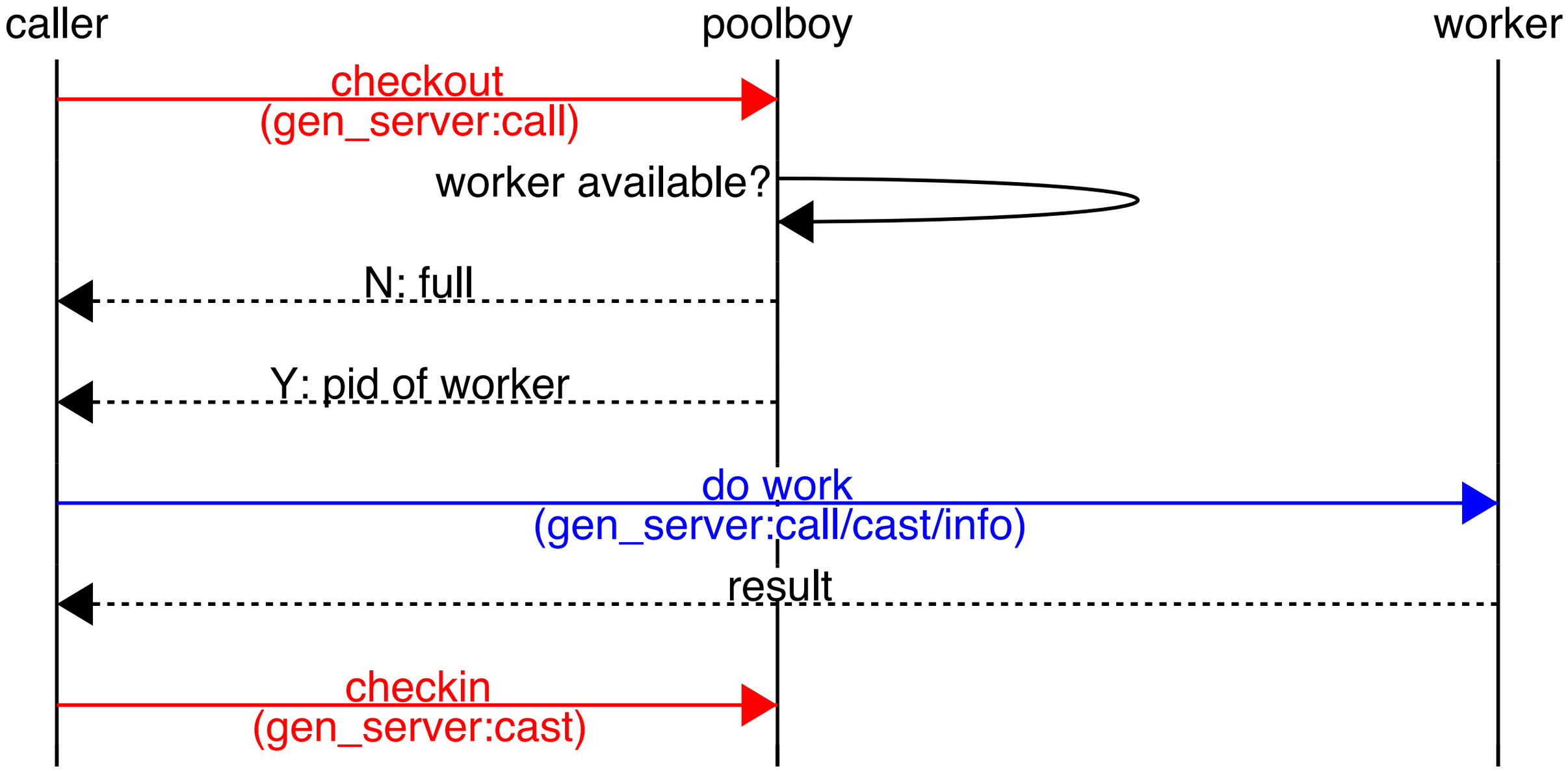
- Most popular based on use in many other packages
- Small - 306 lines of code, 672 lines of tests
- Can queue or fast fail (queue can be lifo or fifo)
- Limited support variable sizing and auto-sizing (via size and overflow)
- Can store any pid (start_link/1 which returns a pid is the only requirement on a worker)

Using poolboy

```
init ([_MinPool, MaxPool]) ->
  { ok,
    { {one_for_one, 10, 10},
      [ { ?POOL_ID,
          {poolboy, start_link,
           [ [{name, {local, ?POOL_ID}},
              {worker_module, pt_baseline_worker},
              {size, MaxPool}, {max_overflow, 0} ],
             WorkerInitArgs ]},
          permanent, 5000, worker, [poolboy] } ] } }.

do (N, Data) ->
  case poolboy:checkout (?POOL_ID, false) of
    full -> {error, busy};
    Worker ->
      Res = pt_baseline_worker:do (Worker, N, Data),
      poolboy:checkin (?POOL_ID, Worker), Res
  end.
```

Call Details poolboy



Overview - pooler

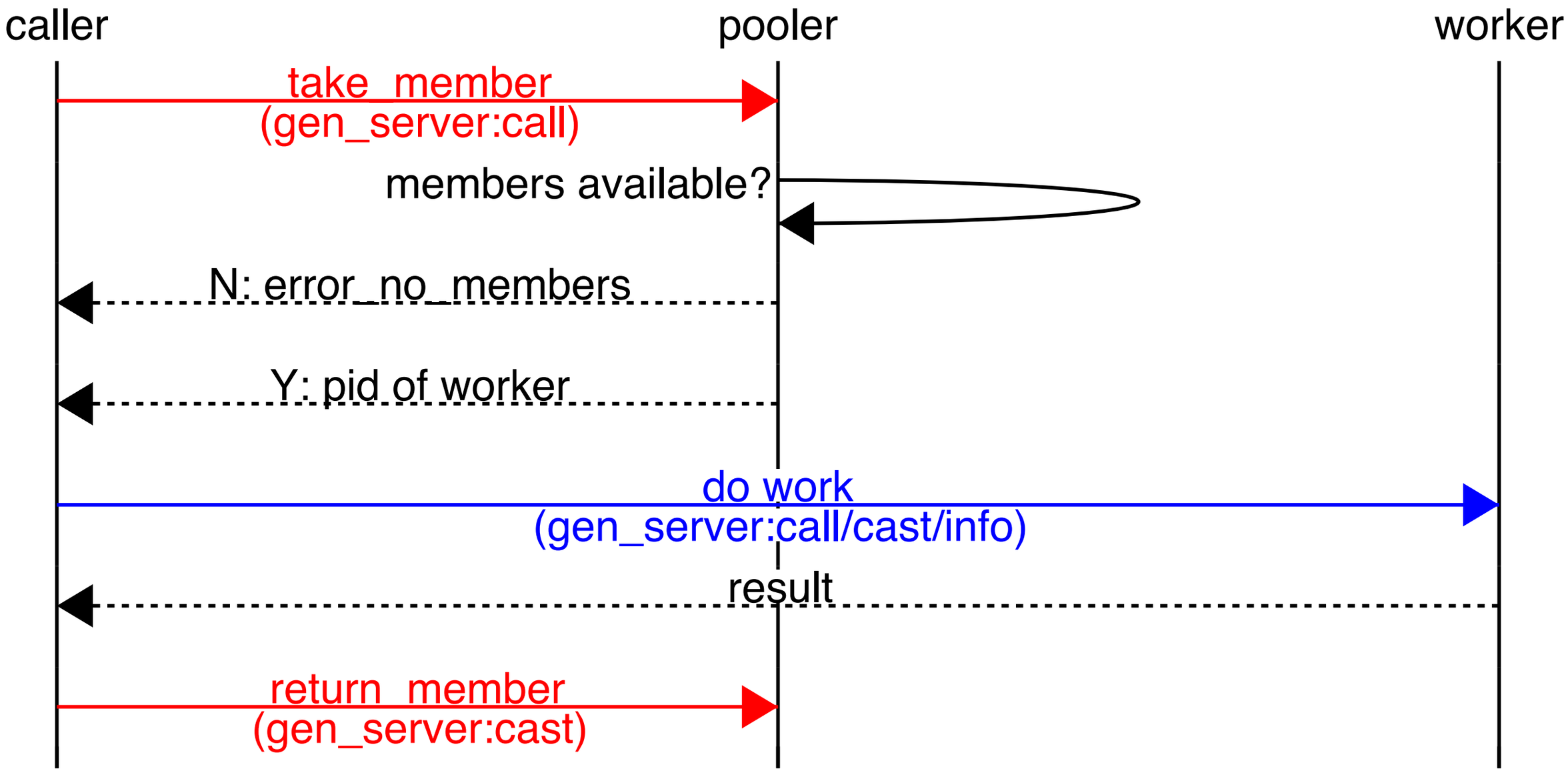
- Complicated OTP supervision tree
- Large - 841 lines of code, 1060 lines of tests
- Unique: supports groups of pools using pg2
- Can queue or fast fail
- Supports variable sizing and auto-sizing based on workers' age
  - culling is a little noisy because of OTP logging
- Can store any pid

Using pooler

```
init ([_MinPool, MaxPool]) ->
  pooler:new_pool (
     [{name, pt_pooler_pool},
      {max_count, MaxPool},
      {init_count, MaxPool},
      {max_age, {60, min}},
       {start_mfa,
        {pt_baseline_worker, start_link, WorkerInitArgs}}],
  {ok, #state {}}.

do (N, Data) ->
  case pooler:take_member (pt_pooler_pool) of
    error_no_members -> {error, busy};
    P -> Res = pt_baseline_worker:do (P, N, Data),
         pooler:return_member (pt_pooler_pool, P, ok),
         Res
  end.
```

Call Details pooler



Overview - gen_server_pool

- Unique: masquerades as worker, so extremely easy to integrate
- Medium - 470 lines of code, 0 lines of test code
- Can queue or fast fail (or both since queue size can be limited)
- Supports variable sizing and auto-sizing based on age or idle
- gen_server pids only

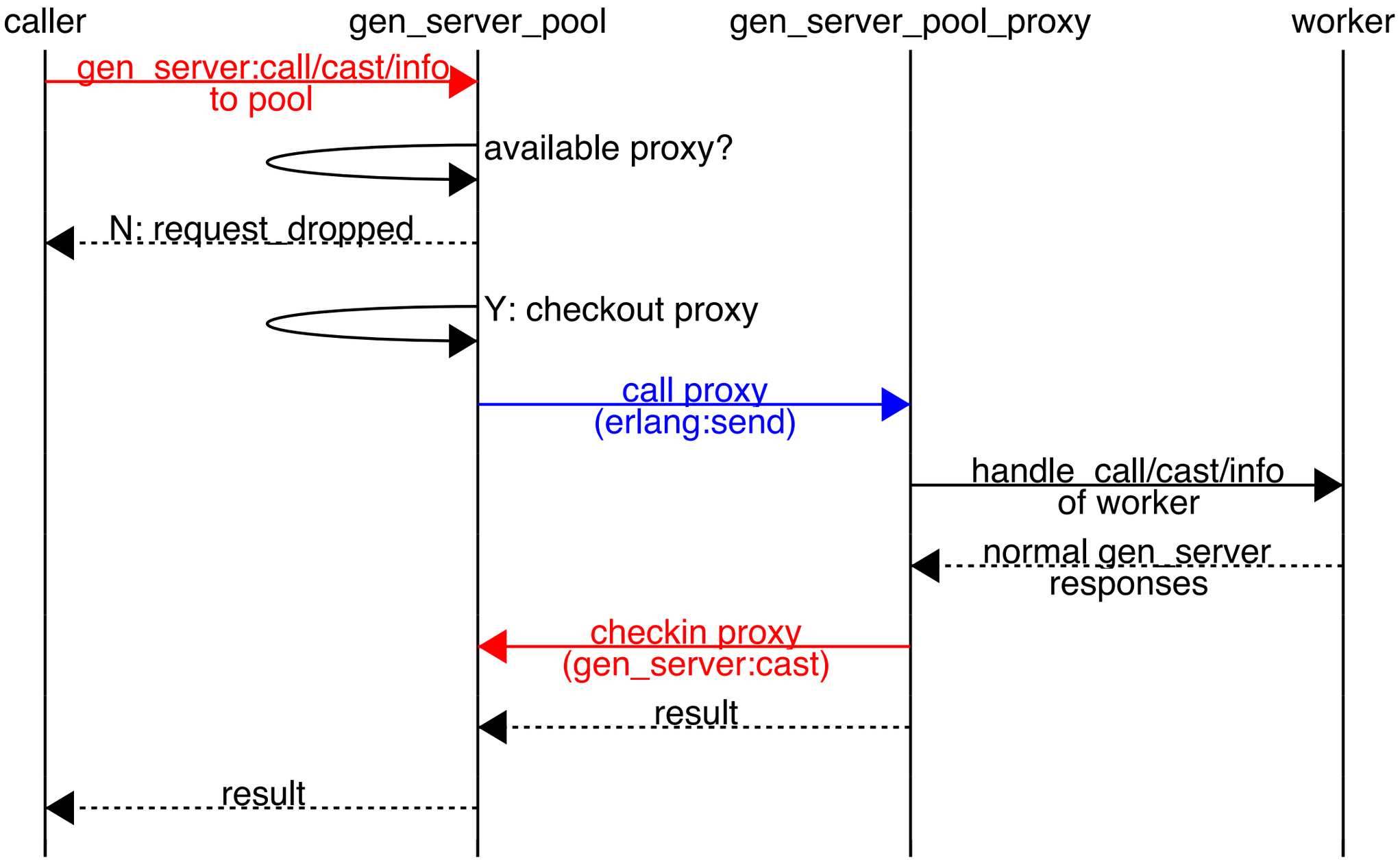
Using gen_server_pool

```
start_link (_MinPool, MaxPool) ->
  PoolOptions =
    [ { min_pool_size, MaxPool },
      { max_pool_size, MaxPool },
      { idle_timeout, 60 }, % seconds
      { max_worker_age, 60 }, % seconds
      { max_queue, 0 },
      { mondemand, false } ],

   gen_server_pool:start_link (
    {local, ?POOL_ID}, pt_baseline_worker,
    WorkerInitArgs, [], PoolOptions).

do (N, Data) ->
  case pt_baseline_worker:do (?POOL_ID, N, Data) of
    {error, request_dropped} -> {error, busy};
    R -> R
  end.
```

Call - gen_server_pool



Overview - dispcount

- Unique stochastic based selection using ETS table or named processes
- Small - 297 lines of code, 361 lines of tests
- Fast fail only (you can yield and retry if you want)
- Fixed number of Resources
- Can store any sort of resource
    - Storing of pid actually results in extra process hops

Using dispcount

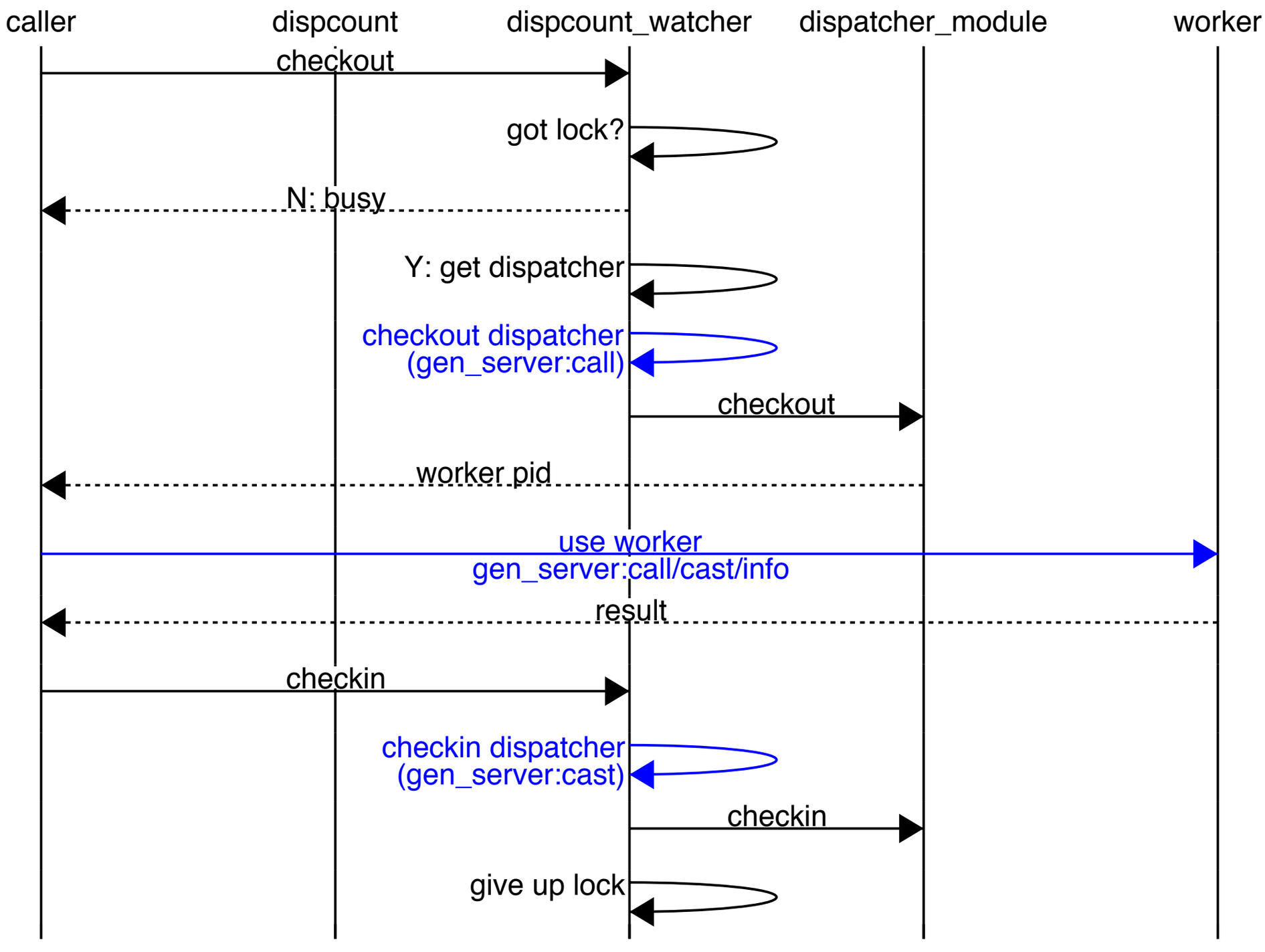
```
init ([_MinPool, MaxPool]) ->
  ok = dispcount:start_dispatch (
         ?POOL_ID, {pt_dispcount_dispatch, WorkerInitArgs},
         [{restart, permanent}, {shutdown, 4000},
           {maxr, 10}, {maxt, 60}, {resources, MaxPool}]
       ),
  {ok, Info} = dispcount:dispatcher_info (?POOL_ID),
  mochiglobal:put (?MOCHIGLOBAL_ID, Info),
  { ok, #state { info = Info } }.

do (N, Data) ->
  PoolInfo = mochiglobal:get (?MOCHIGLOBAL_ID),
  case dispcount:checkout (PoolInfo) of
    {error, busy} -> {error, busy};
    {ok, CheckinReference, Pid} ->
      Res = pt_baseline_worker:do (Pid, N, Data),
      dispcount:checkin(PoolInfo, CheckinReference, Pid),
      Res
  end.
```

Using dispcount (cont.)

```
-behaviour (dispcount).
-record (state, {pid, given=false, args}).
init(WorkerInitArgs) ->
  {ok, P} = pt_baseline_worker:start_link (WorkerInitArgs),
  {ok, #state {pid = P, args = WorkerInitArgs}}.
checkout(_From, State = #state {given=true}) ->
  {error, busy, State};
checkout(_From, State = #state {pid=Pid}) ->
  {ok, Pid, State#state {given=true}}.
checkin(Pid, State = #state {pid=Pid, given=true}) ->
  {ok, State#state {given=false}};
checkin(_Pid, State) ->
  {ignore, State}.
dead(State = #state {args = WorkerInitArgs}) ->
  {ok, P} = pt_baseline_worker:start_link (WorkerInitArgs),
  %% lost resource so start a new one
  {ok, State#state {pid=P, given=false}}.
```

Call - dispcount



Overview - gproc

- Really talking about gproc_pool
- Offers just the dispatching method (but offers several)
- gproc must be used for managing processes
- gproc_pool small - 558 lines of code, 98 lines of tests (but gproc is XL - 4090 lines of code 1788 lines of tests).
- Active queuing (via loop over erlang:yield/0) or fail fast
- Does not support sizing
- Stores pids only
- Dispatch is purely ETS based
- Requires modifications to your worker process

Using gproc

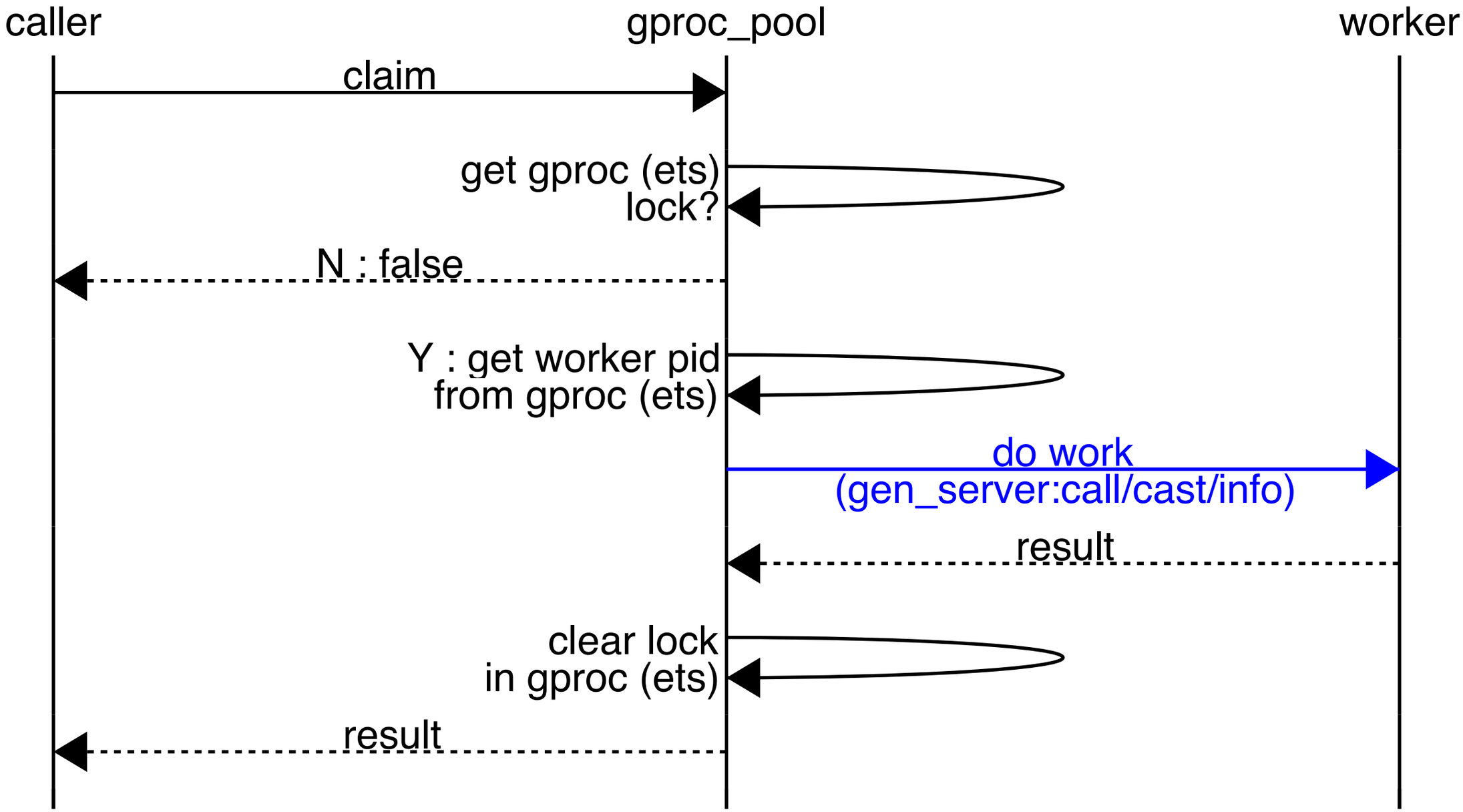
```
init ([_MinPool, MaxPool]) ->
  ok = gproc_pool:new (?POOL_ID, claim, []),
  { ok, { {one_for_one, 10, 10},
       [ begin
           WorkerName = {?POOL_ID, N},
           gproc_pool:add_worker (?POOL_ID, WorkerName),
           { WorkerName,
             {pt_gproc_worker, start_link, [?POOL_ID, WorkerName]},
             transient, 2000, worker, [pt_gproc_worker] }
         end
         || N <- lists:seq (1, MaxPool) ] } }.

do (N, Data) ->
  case gproc_pool:claim (?POOL_ID,
    fun (_,Pid) -> pt_gproc_worker:do (Pid, N, Data) end) of
      false -> {error, busy};
      {true, Res} -> Res
  end.
```

Using gproc (cont.)

```
% gen_server init function
init ([PoolName, Name]) ->
  % ensure terminate is called
process_flag( trap_exit, true ),
gproc_pool:connect_worker (PoolName, Name),
{ok, #state {supervisor = PoolName, name = Name}}.
```

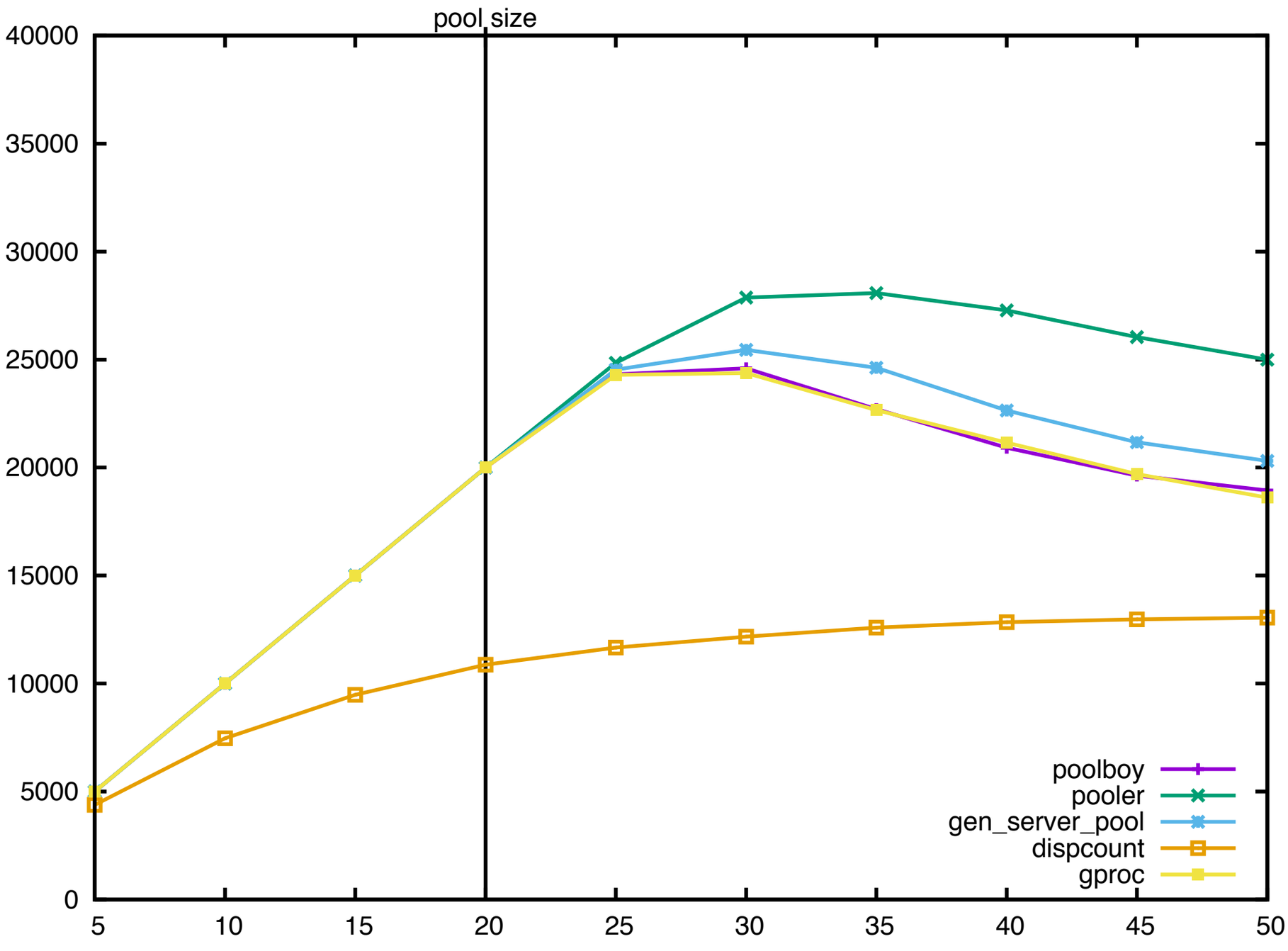
Call - gproc


caller
gproc_pool
worker
claim
get gproc (ets)
lock?
N : false
Y : get worker pid
from gproc (ets)
do work
(gen_server:call/cast/info)
result
clear lock
in gproc (ets)
result

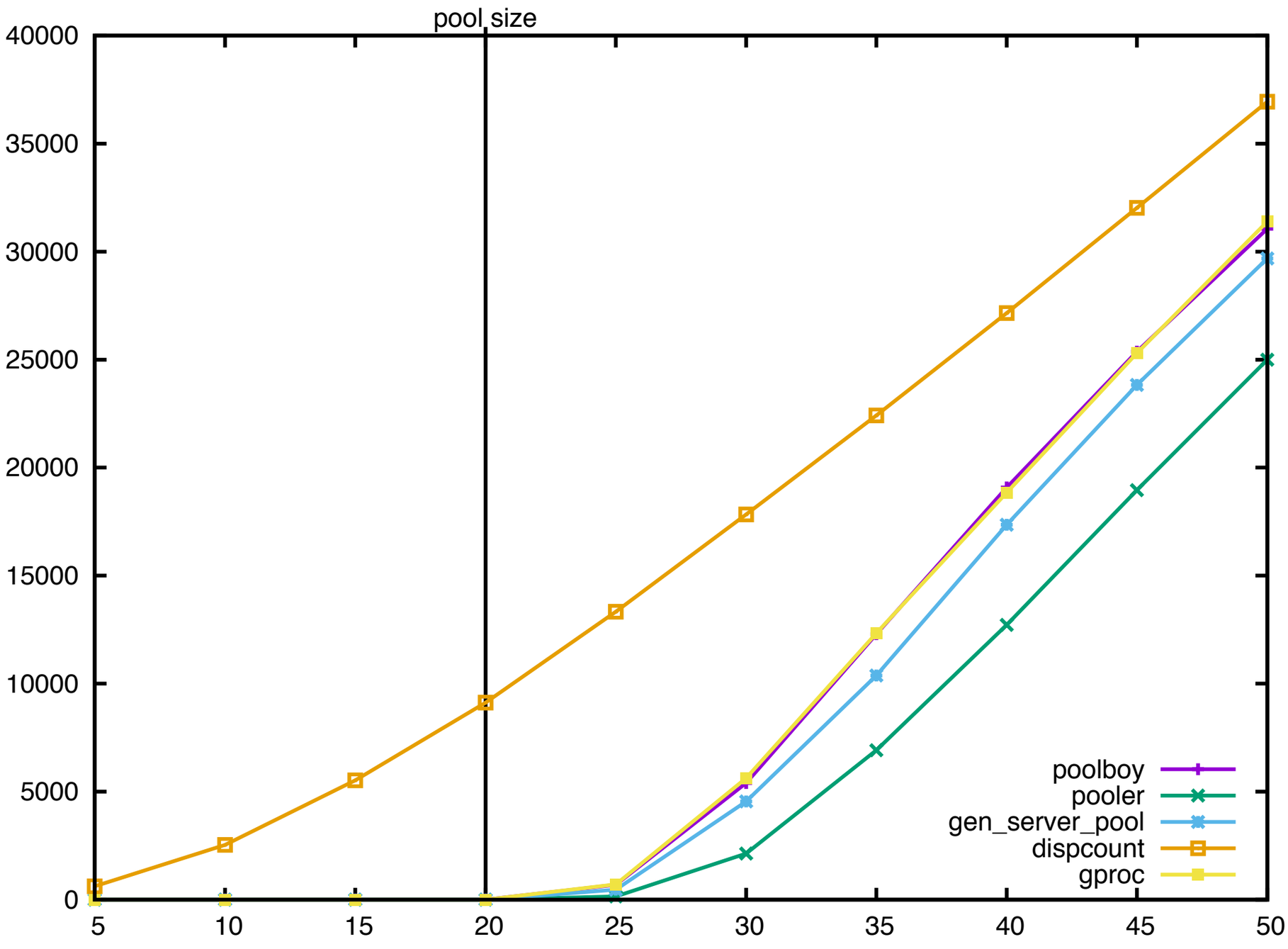
Comparative Peformance

- Using most common settings
  - fixed size pool (20 processes)
  - fail fast config
- Spawn a number of callers, each call
  - gets a worker, does work (sleep of 5 ms), sleeps for a small random amount of time (1-5 ms), repeats for some number of iterations
- Measure good vs. busy responses, and min/avg/max time

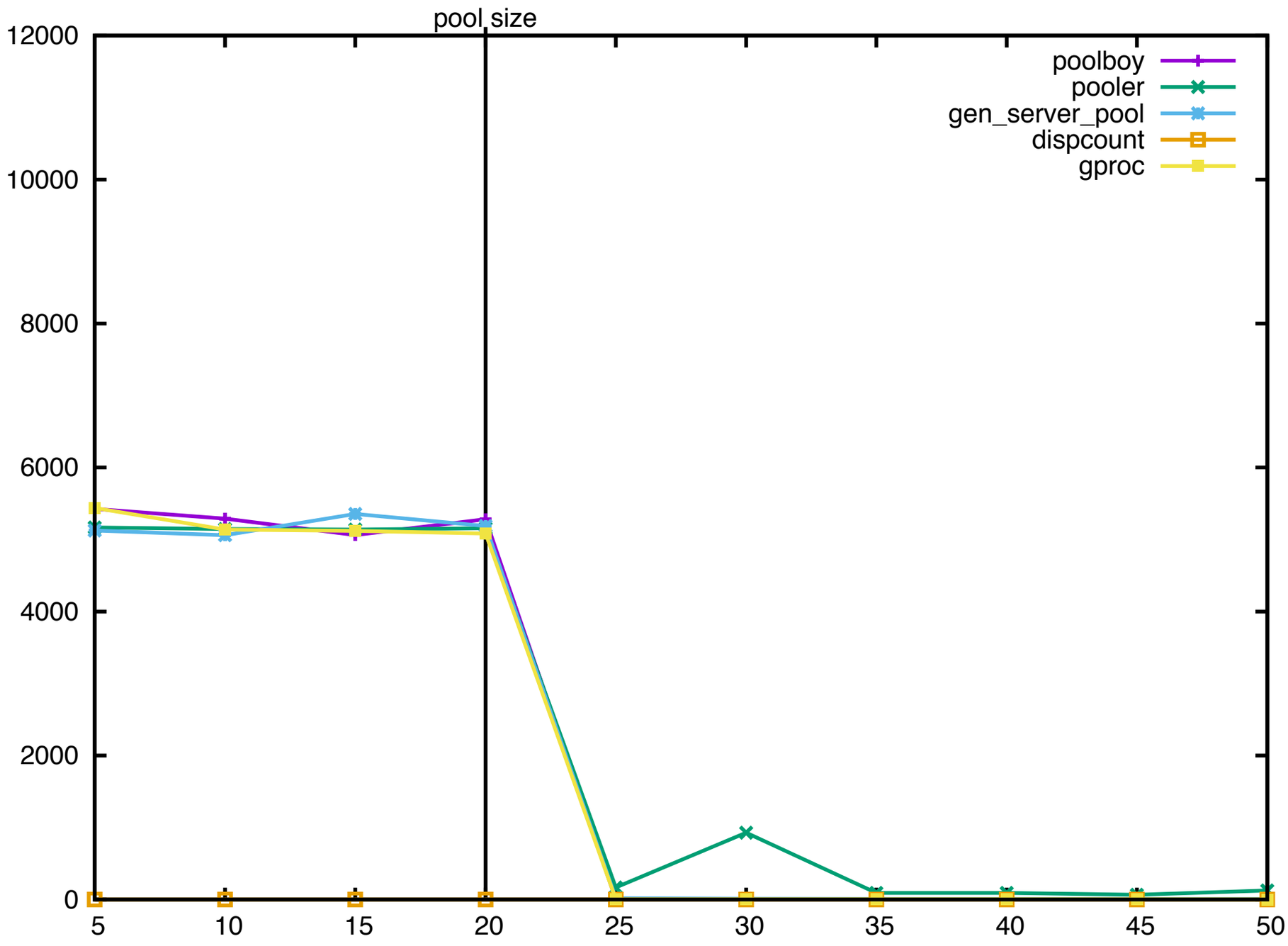
Good Results



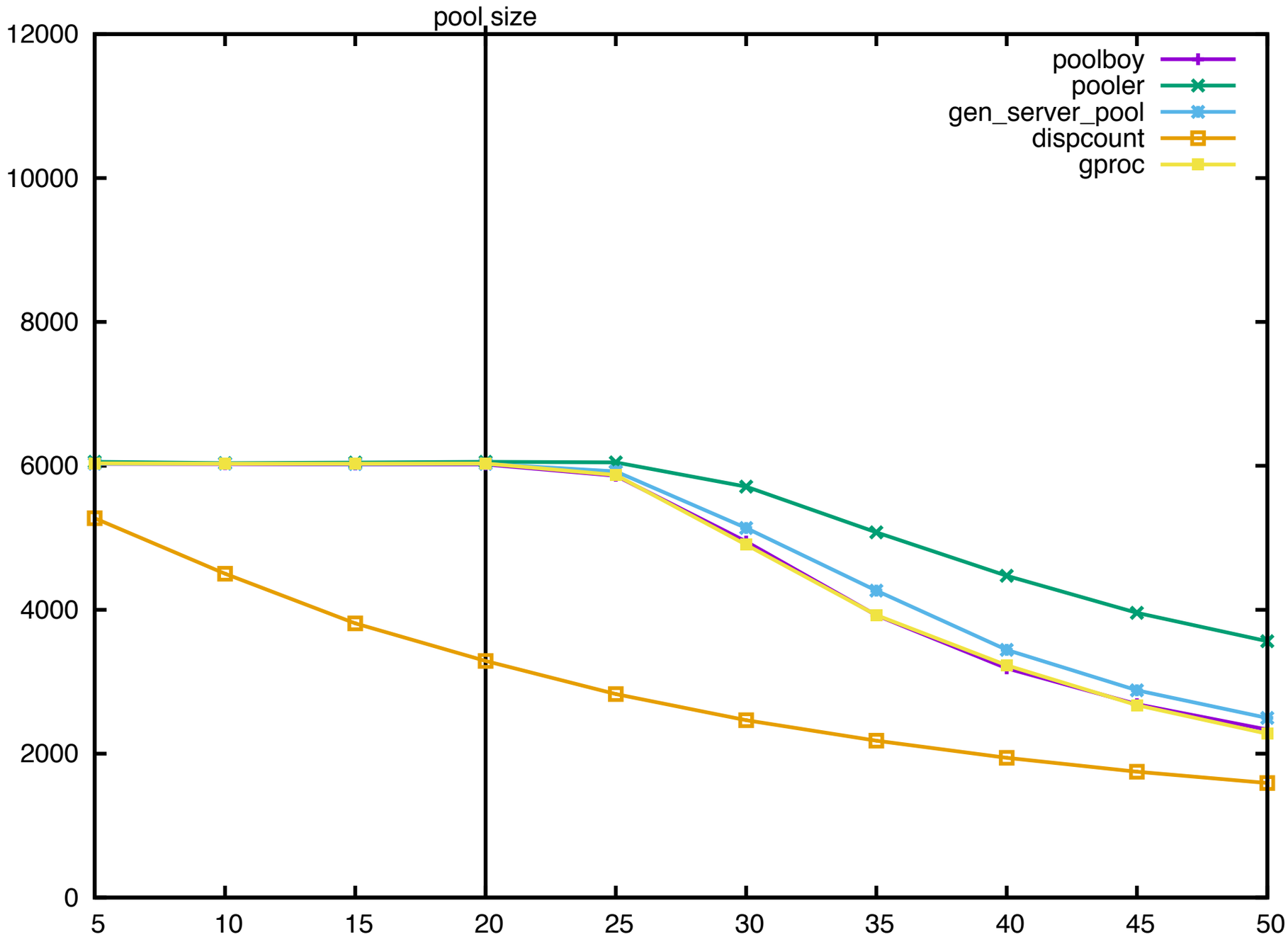
Busy Results



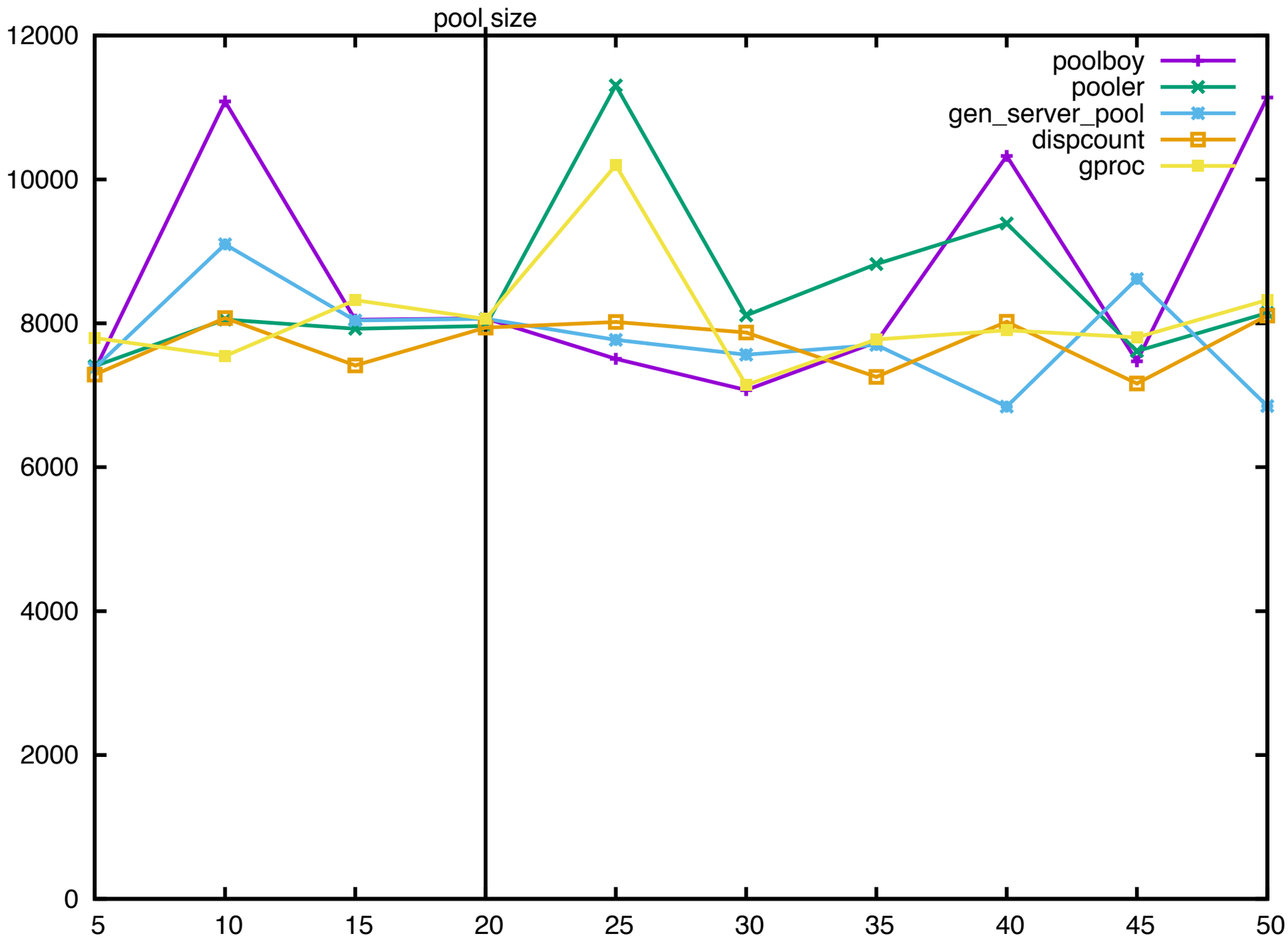
Min Time


pool size
12000
10000
8000
6000
4000
2000
0
5
10
15
20
25
30
35
40
45
50
poolboy
pooler
gen_server_pool
dispcount
gproc

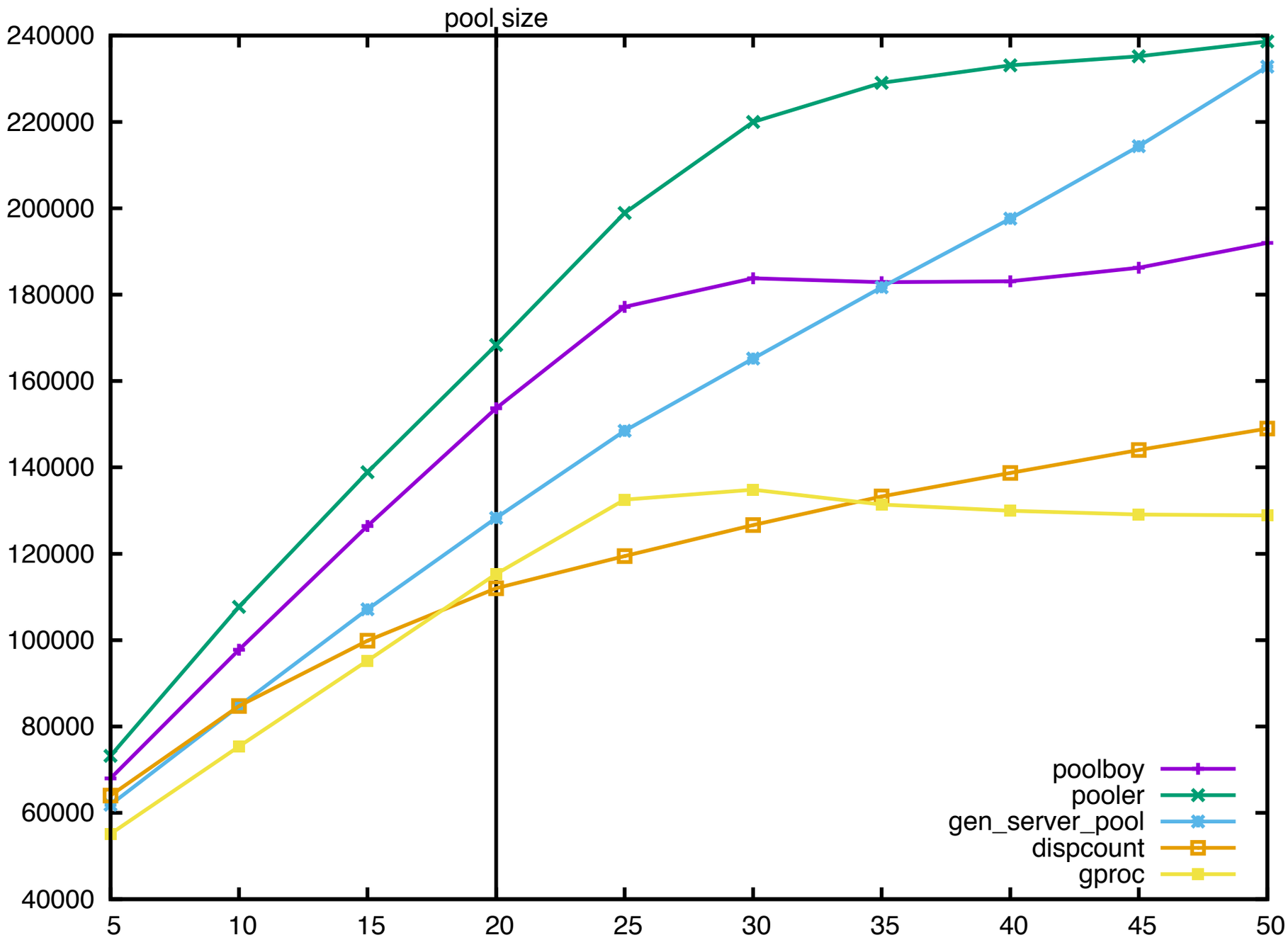
Average Time



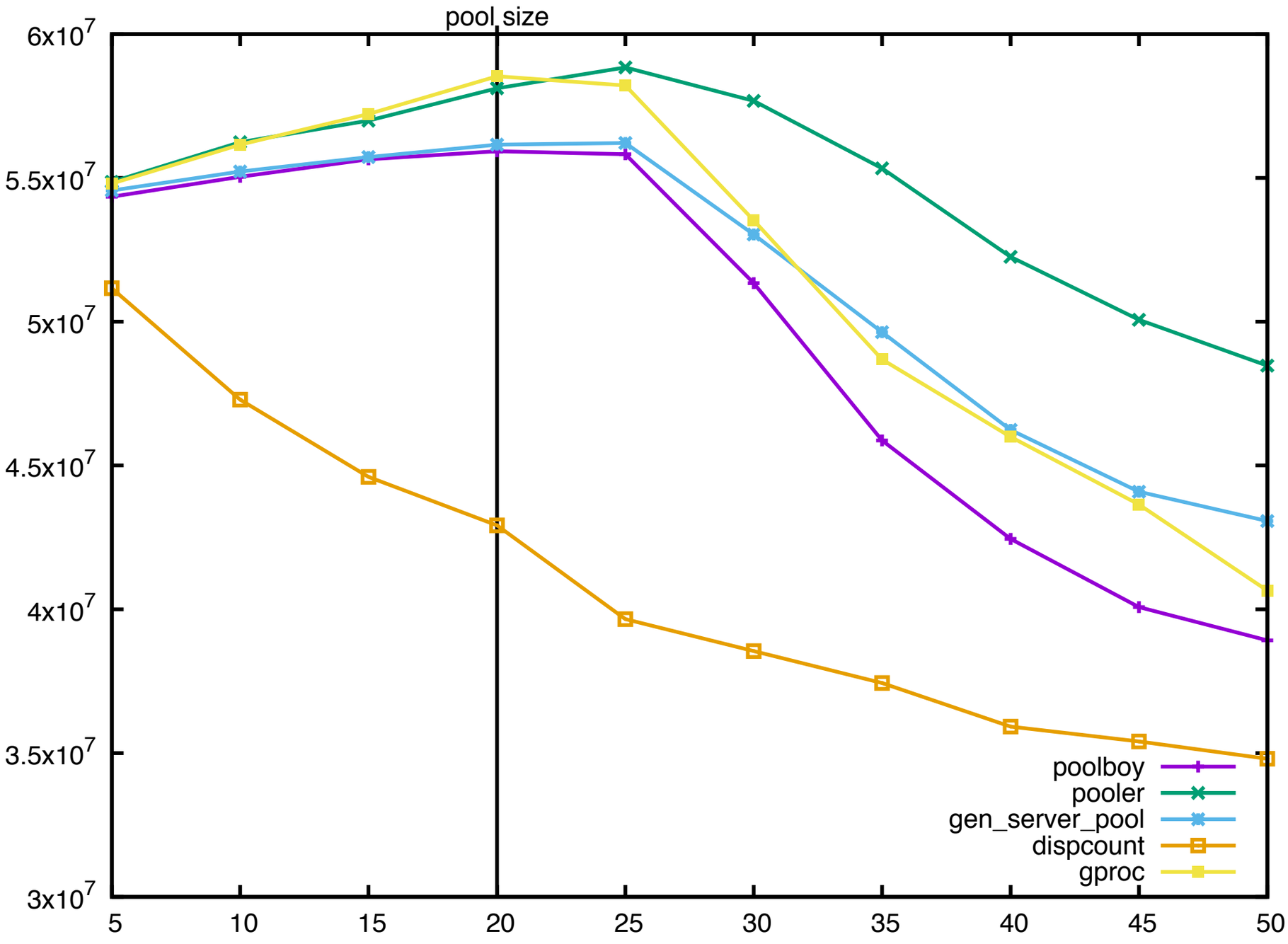
Max Time


pool size
12000
10000
8000
6000
4000
2000
0
5
10
15
20
25
30
35
40
45
50
poolboy
pooler
gen_server_pool
dispcount
gproc

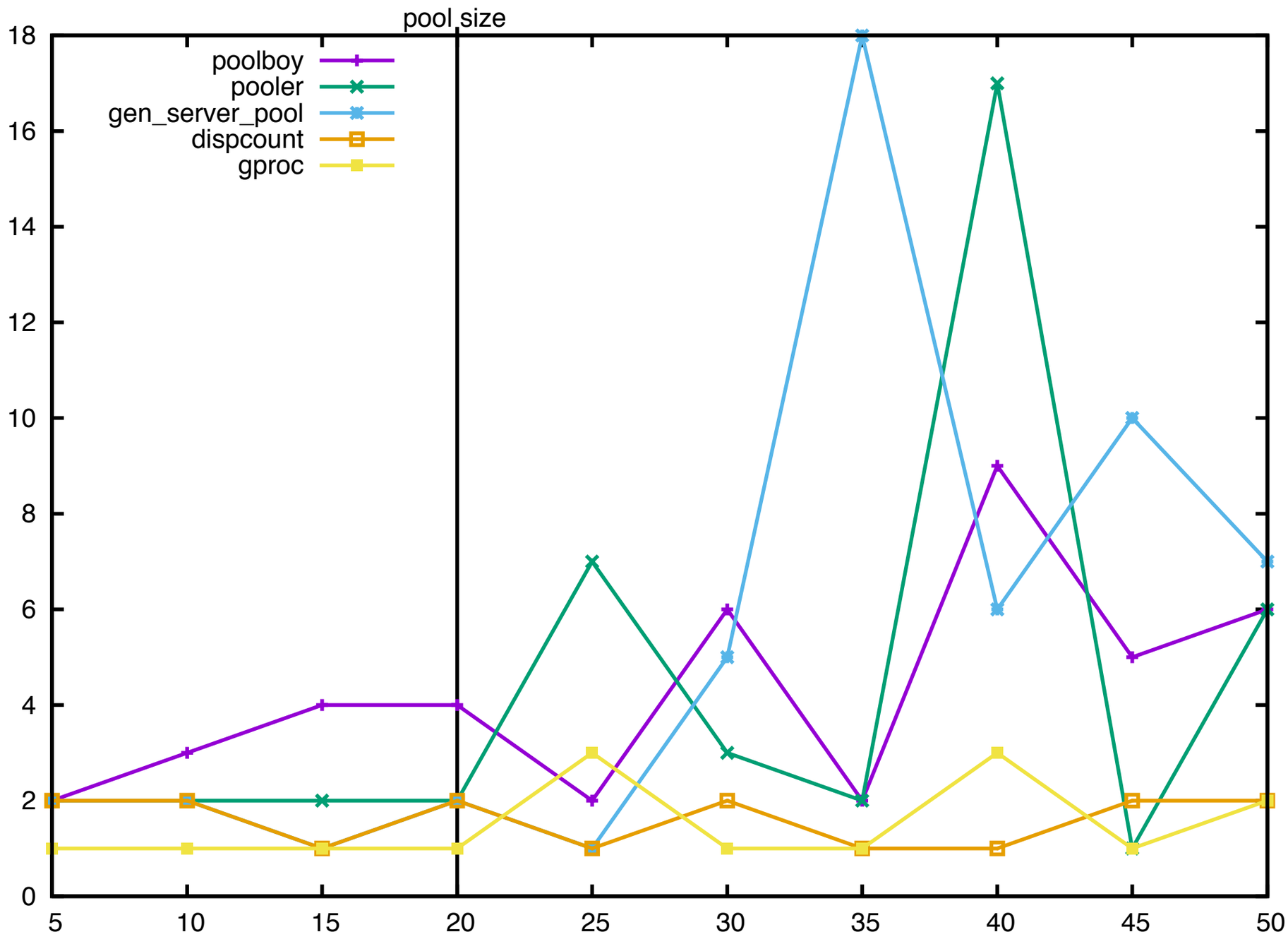
Context Switches



Reductions


pool size
6x10^7
5.5x10^7
5x10^7
4.5x10^7
4x10^7
3.5x10^7
3x10^7
5
10
15
20
25
30
35
40
45
50
poolboy
pooler
gen_server_pool
dispcount
gproc

Max Message Queue Size



General recommendations

- poolboy is popular and widely used, it's easy enough to integrate with, and you'll likely find lots of help with issues, but beware it may have issues with load

- pooler is a close second, but it's completeness with regards to OTP might make it harder to work with, but beware it may have issues with load

- gen_server_pool is super easy to integrate with, production hardened, but not widely used and may have issues with load

- dispcount is good for the reasons stated on its github page, in short if you you know you'll be overdriving a limited set of resources and fail fast is the behavior you want it'll be the fastest at doing that

- gproc seems like it should deal the best with load, but requires you to write your own process management as well as requires changes to an existing worker

When not to pool

- A single gen_server never becomes a bottleneck, it's so fast you never notice it backing up.

- Large fixed data structures can be compiled into modules and shared with little cost by many processes (you may even be able to hot load updates).

- Passing around ports is often faster and easier than wrapping a process around them and pooling them, so if you already have a way to reuse acceptors on your front end you might be able to reuse a port on the backend.

Links & QA

- Example Code + Notes
  - https://github.com/djnym/erlang-pool-research
- Contact info
  - anthony.molinaro@openx.com
- Thanks to OpenX for giving me the time to research this talk. If you want to talk about opportunities to work in Erlang everyday come talk to me.