Designing for Scale

Knut Nesheim @knutin
Paolo Negri @hungryblank

wooga
world of gaming

About this talk

2 developers and erlang
vs.
1 million daily users

wooga
world of gaming

Social Games

Flash client (game)

HTTP API

wooga world of gaming

Social Games

Flash client

- Game actions need to be persisted and validated
- 1 API call every 2 secs

wooga world of gaming

Social Games

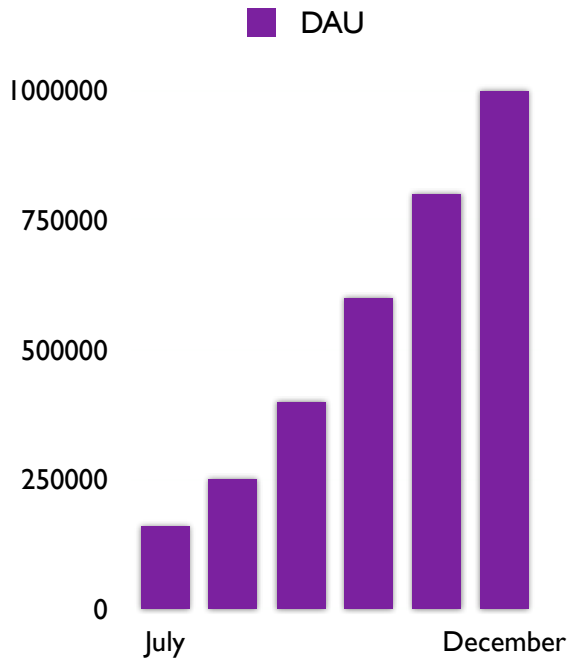
HTTP API

- @ 1 000 000 daily users
- 5000 HTTP reqs/sec
- more than 90% writes

wooga
world of gaming

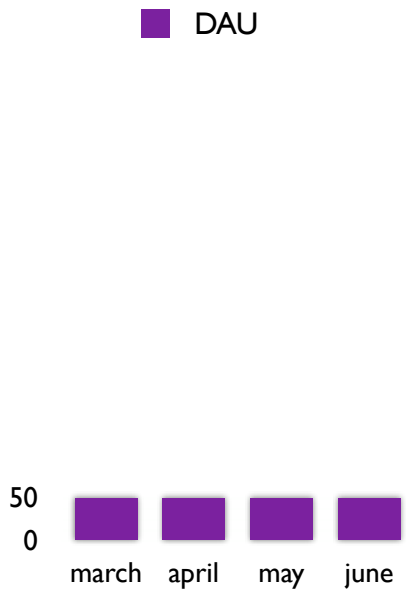
Users we expect

"Monster World"
daily users
july - december 2010



Users we have

New game
daily users
march - june 2011



What to do?

1 Simulate users

Simulating users

- Must not be too synthetic (like apachebench)
- Must look like a meaningful game session
- Users must come online at a given rate and play

Tsung

- Multi protocol (HTTP, XMPP) benchmarking tool
- Able to test non trivial call sequences
- Can actually simulate a scripted gaming session

http://tsung.erlang-projects.org/

Tsung - configuration

Fixed content Dynamic parameter

```
<request subst="true">
<http url="http://server.wooga.com/users/%
%ts_user_server:get_unique_id%%/resources/column/5/
row/14?%%_routing_key%%"
method="POST" contents='{"parameter1":"value1"}'>
</http>
</request>
```

http://tsung.erlang-projects.org/

Tsung - configuration

- Not something you fancy writing
- We're in development, calls change and we constantly add new calls
- A session might contain hundreds of requests
- All the calls must refer to a consistent game state

http://tsung.erlang-projects.org/

Tsung - configuration

- From our ruby test code

```
user.resources(:column => 5, :row => 14)
```

- Same as

```
<request subst="true">
<http url="http://server.wooga.com/users/%
%ts_user_server:get_unique_id%%/resources/column/5/
row/14?%%_routing_key%%"
method="POST" contents='{"parameter1":"value1"}'>
</http>
</request>
```

http://tsung.erlang-projects.org/

Tsung - configuration

- Session
  - requests
- Arrival phase
  - duration
  - arrival rate

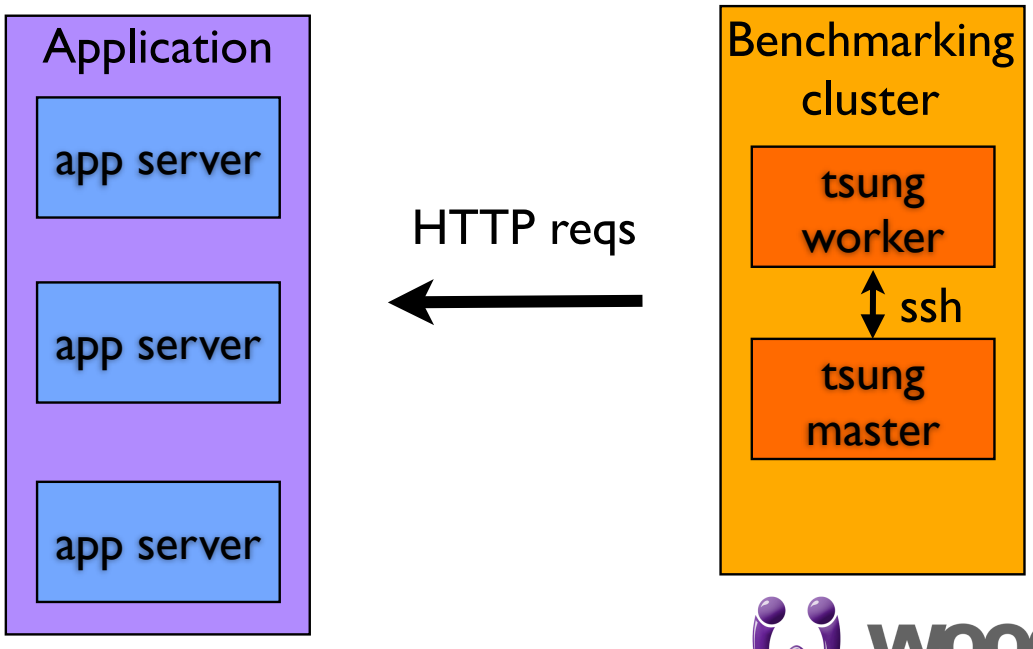
A session is a group of requests

Sessions arrive in phases with a specific arrival rate

http://tsung.erlang-projects.org/

wooga
world of gaming

Tsung - setup



http://tsung.erlang-projects.org/

wooga
world of gaming

Tsung

- Generates ~ 2500 reqs/sec on AWS m1.large
- Flexible but hard to extend
- Code base rather obscure

http://tsung.erlang-projects.org/

What to do?

2 Collect metrics

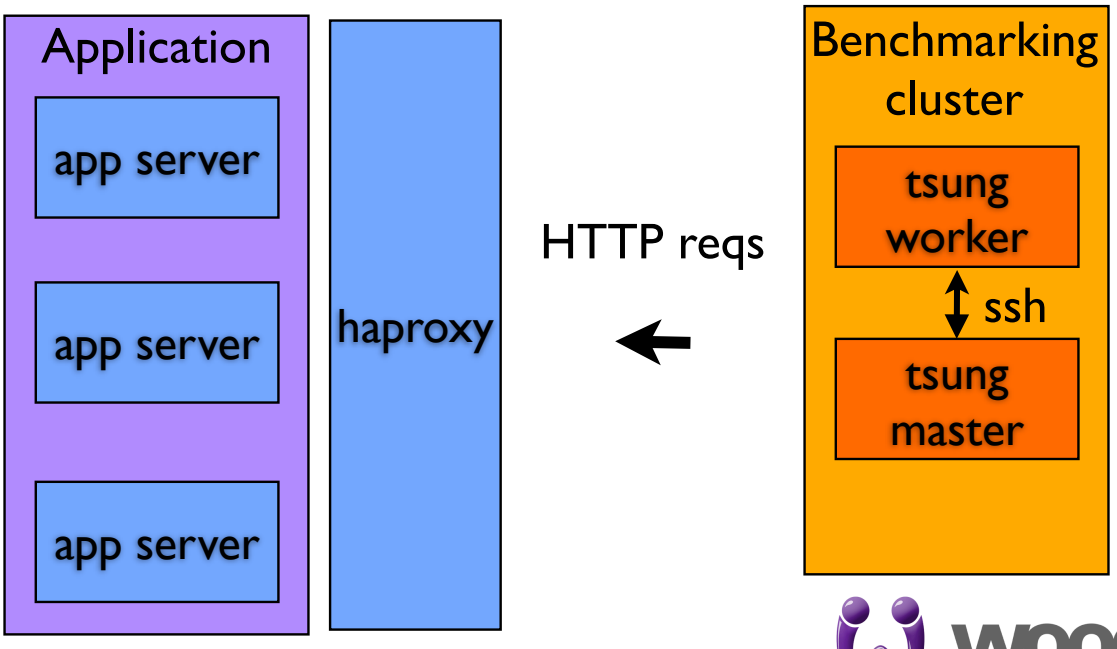
Tsung-metrics

- Tsung collects measures and provides reports
- But these measure include tsung network/ cpu congestion itself
- Tsung machines aren't a good point of view

http://tsung.erlang-projects.org/

wooga
world of gaming

HAproxy



wooga
world of gaming

HAproxy

"The Reliable, High Performance TCP/ HTTP Load Balancer"

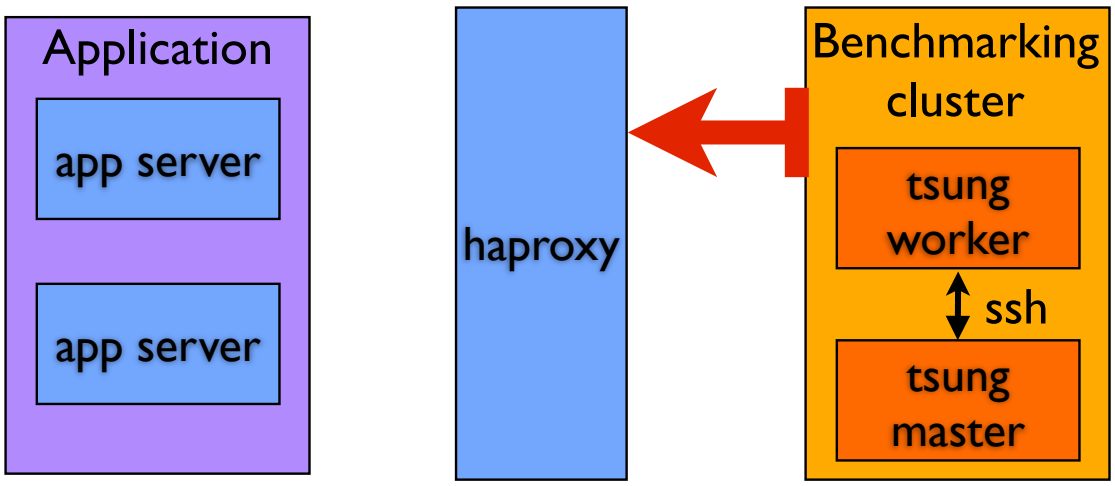
- Placed in front of http servers
- Load balancing
- Fail over

HAproxy - syslog

- Easy to setup
- Efficient (UDP)
- Provides 5 timings per each request

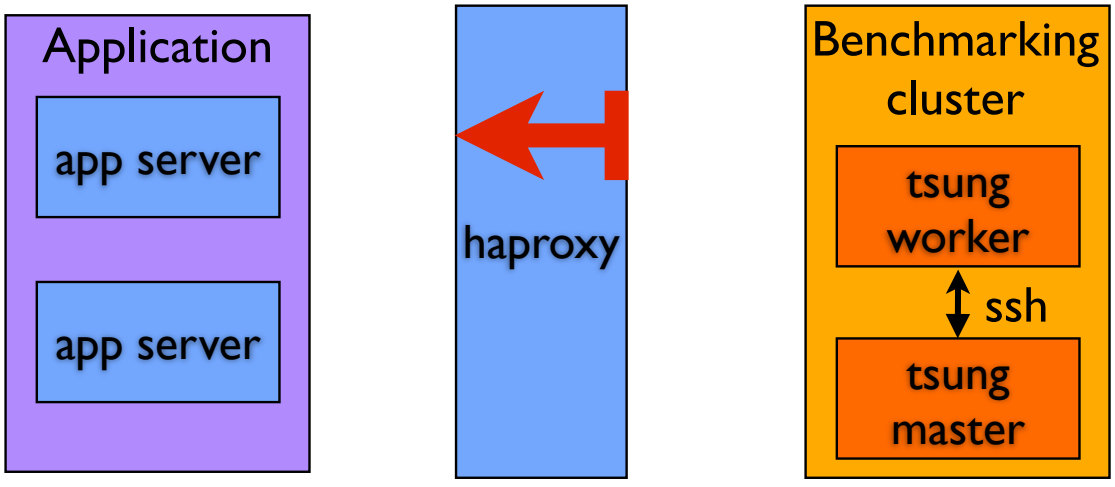
HAproxy

- Time to receive request from client



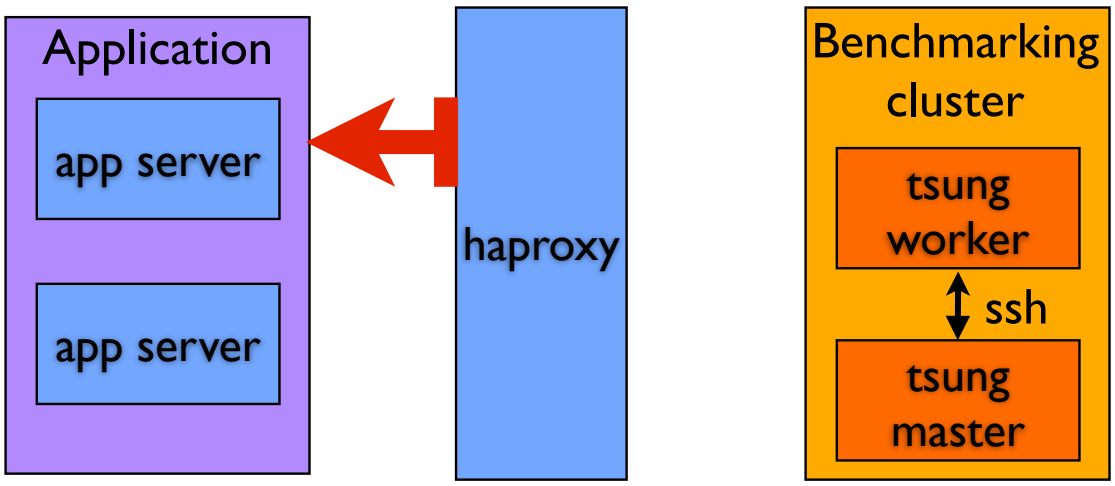
HAproxy

- Time spent in HAproxy queue



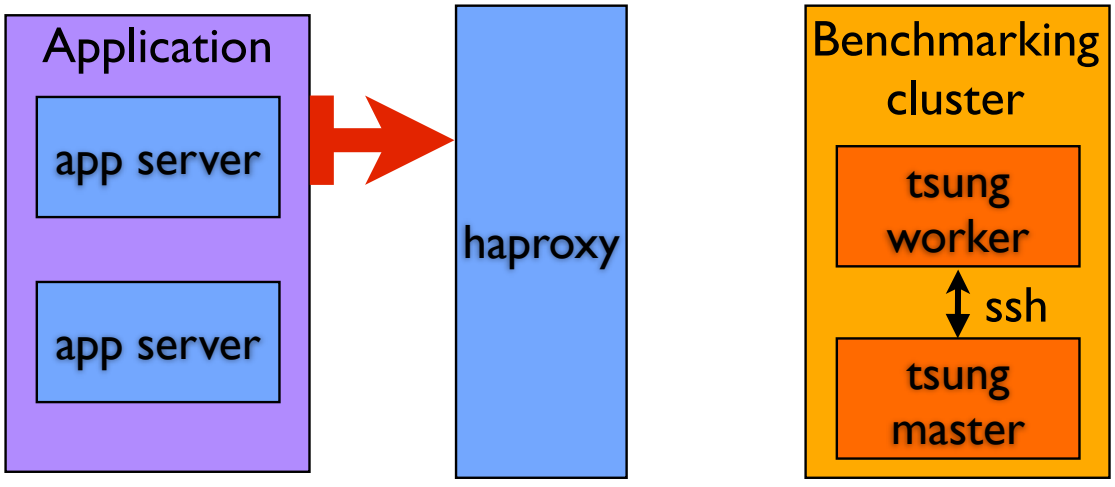
HAproxy

- Time to connect to the server



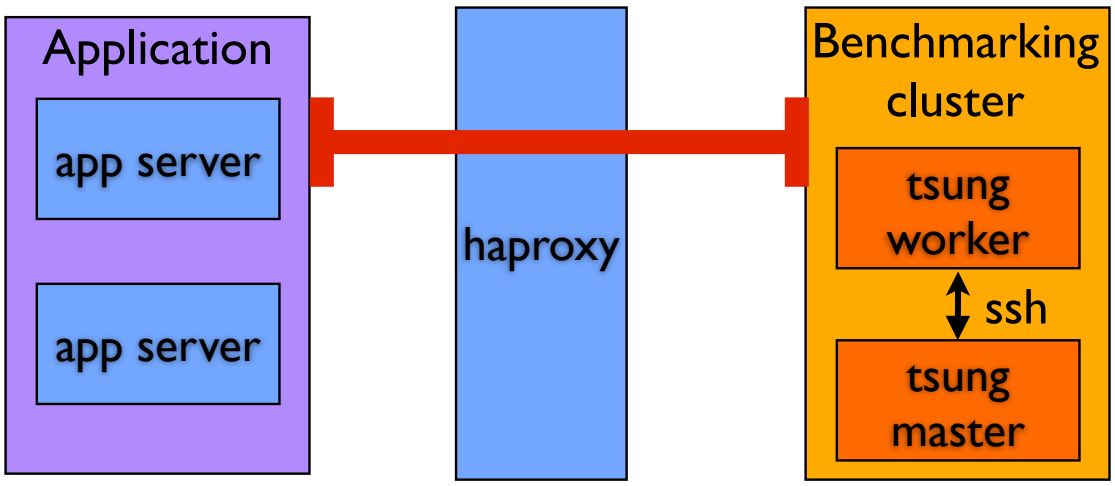
HAproxy

- Time to receive response headers from server



HAproxy

- Total session duration time



HAproxy - syslog

- Application urls identify directly server call
- Application urls are easy to parse
- Processing haproxy syslog gives per call metric

wooga
world of gaming

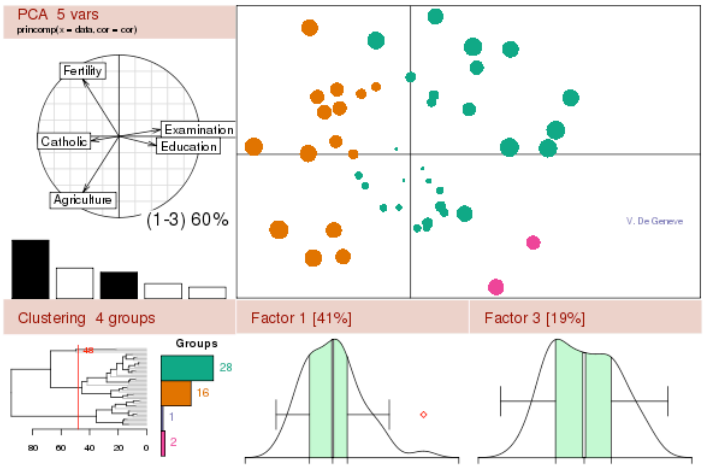
What to do?

3 Understand metrics

Reading/aggregating metrics

- Python to parse/normalize syslog
- R language to analyze/visualize data
- R language console to interactively explore benchmarking results

R is a free software environment for statistical computing and graphics.



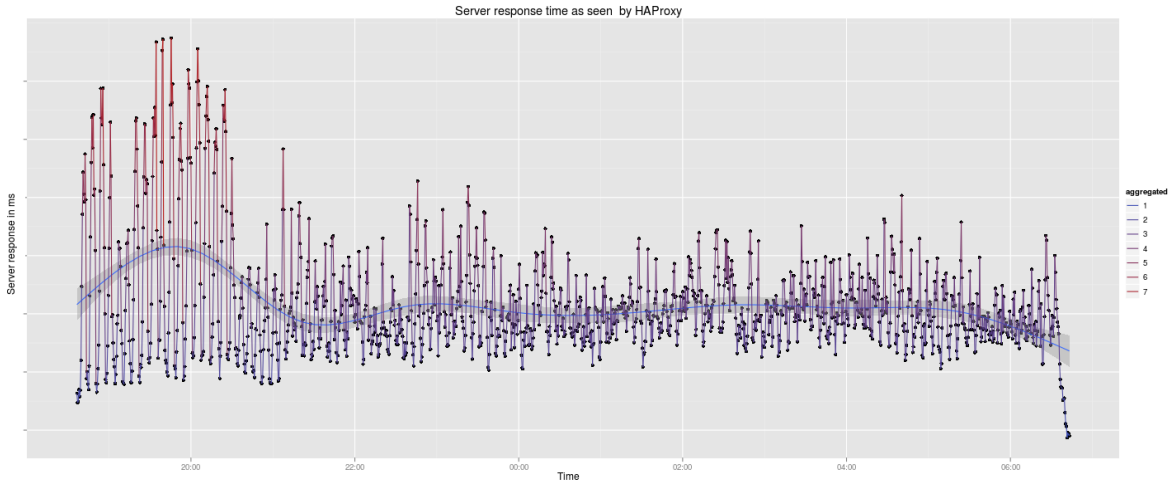
wooga
world of gaming

What you get

- Aggregate performance levels (throughput, latency)
- Detailed performance per call type
- Statistical analysis (outliers, trends, regression, correlation, frequency, standard deviation)

wooga
world of gaming

What you get



wooga
world of gaming

What to do?

4 go deeper

wooga
world of gaming

Digging into the data

- From HAproxy log analisys one call emerged as exceptionally slow
- Using eprof we were able to determine that most of the time was spent in a redis query fetching many keys (MGET)

Tracing erldis query

- More than 60% of runtime is spent manipulating the socket
- gen_tcp:recv/2 is the culprit
- But why is it called so many times?

```
prim_inet:enc_opt/1                    1000   2.14   502  [      0.50]
prim_inet:type_value/3                 1000   2.47   578  [      0.58]
inet:setopts/2                         1501   2.75   643  [      0.43]
prim_inet:encode_opt_val/1             1501   2.86   670  [      0.45]
erldis_client:trim2/1                  2002   3.62   848  [      0.42]
prim_inet:ctl_cmd/3                    2001   4.03   944  [      0.47]
prim_inet:enc_opt_val/2                3002   4.51  1056  [      0.35]
erlang:setelement/3                    2018   5.75  1347  [      0.67]
prim_inet:enum_val/2                  11000  12.09  2830  [      0.26]
erlang:port_control/3                  2001  16.20  3794  [      1.90]
```

Understanding the redis protocol

C: LRANGE mylist 0 2

s: *2

s: $5

s: Hello

s: $5

s: World

```
<<"*2\r\n
   $5\r\n
   Hello\r\n
   $5\r\n
   World\r\n">>
```

Understanding erldis

- recv_value/2 is used in the protocol parser to get the next data to parse

```
recv_value(Socket, NBytes) ->
    inet:setopts(Socket, [{packet, 0}]),

    %% Read raw bytes from the socket
    case gen_tcp:recv(Socket, NBytes+2) of
        {ok, Packet} ->
            inet:setopts(Socket, [{packet, line}]),
            trim2(Packet);
        ...
    end.
```

A different approach

- Two ways to use gen_tcp: active or passive
- In passive, use gen_tcp:recv to explicitly ask for data, blocking
- In active, gen_tcp will send the controlling process a message when there is data
- Hybrid: active once

A different approach

- Is active sockets faster?
- Proof-of-concept proved active socket faster
- Change erldis or write a new driver?

A different approach

- Radical change => new driver
- Keep Erldis queuing approach
- Think about error handling from the start
- Use active sockets

A different approach

- Active socket, parse partial replies

```
handle_info({tcp, _, Data}, State) ->
    case eredis_parser:parse(State#state.parserstate, Data) of
        %% Got complete response, return value to client
        {ReturnCode, Value, NewParserState} ->
            reply({ReturnCode Value}, Client)
            ...
            State#state{parser_state = NewParserState};
        %% Parser needs more data, the parser state now contains the
        %% continuation data and we will continue where it stopped
        %% when we receiver more data
        {continue, NewParserState} ->
            State#state{parser_state = NewParserState}
    end.
```

Circuit breaker

- eredis has a simple circuit breaker for when Redis is down/unreachable
- eredis returns immediately to clients if connection is down
- Reconnecting is done outside request/response handling
- Robust handling of errors

Benchmarking eredis

- Redis driver critical for our application
- Must perform well
- Must be stable
- How do we test this?

Basho bench

- Basho produces the Riak KV store
- Basho build a tool to test KV servers
- Basho bench
- We used Basho bench to test eredis

Basho bench

- Create callback module

```
run(get, KeyGen, _ValueGen, Client) ->
    case eredis:q(Client, ["GET", KeyGen()]) of
        {ok, _Value} ->
            {ok, Client};
        ...
    end;

run(put, KeyGen, ValueGen, Client) ->
    case eredis:q(Client, ["SET", KeyGen(), ValueGen()]) of
        {ok, <<"OK">>} ->
            {ok, Client};
        ...
    end.
```

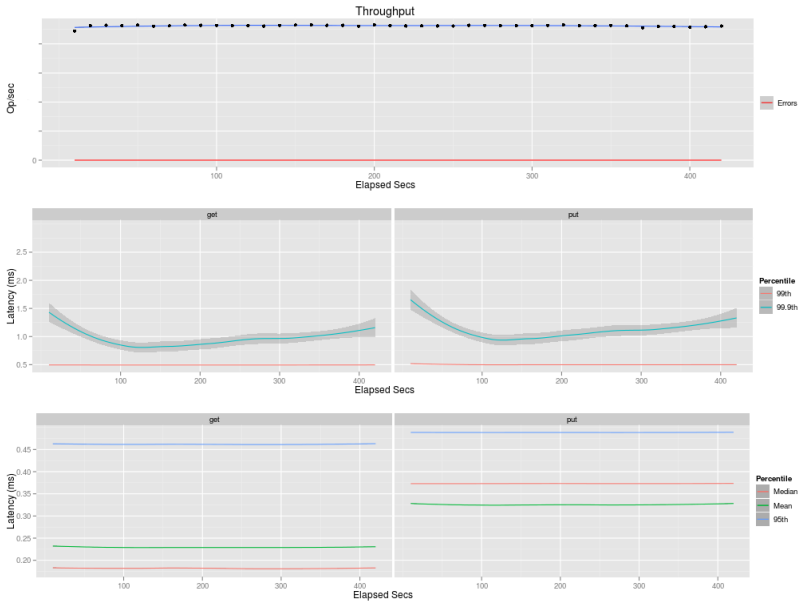
Basho bench

- Configuration term-file

```
{mode, max}.
{duration, 15}.
{concurrent, 30}.
{driver, basho_bench_driver_eredis}.
{code_paths, ["/home/knutin/git/eredis/ebin/"]}.
{operations, [{get,1}, {put,4}]}.
{key_generator, {uniform_int, 10000}}.
{value_generator, {function, basho_bench_driver_eredis, value_gen, []}}.
```

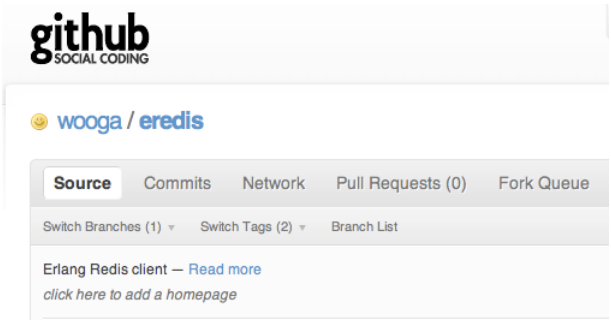
wooga
world of gaming

Basho bench output



wooga
world of gaming

eredis is open source



https://github.com/wooga/eredis

wooga
world of gaming

What to do?

5 measure internals

wooga
world of gaming

Measure internals

HAproxy point of view is valid but how to measure internals of our application, while we are live, without the overhead of tracing?

Think Basho bench

- Basho bench can benchmark a redis driver
- Redis is very fast, 100K ops/sec
- Basho bench overhead is acceptable
- The code is very simple

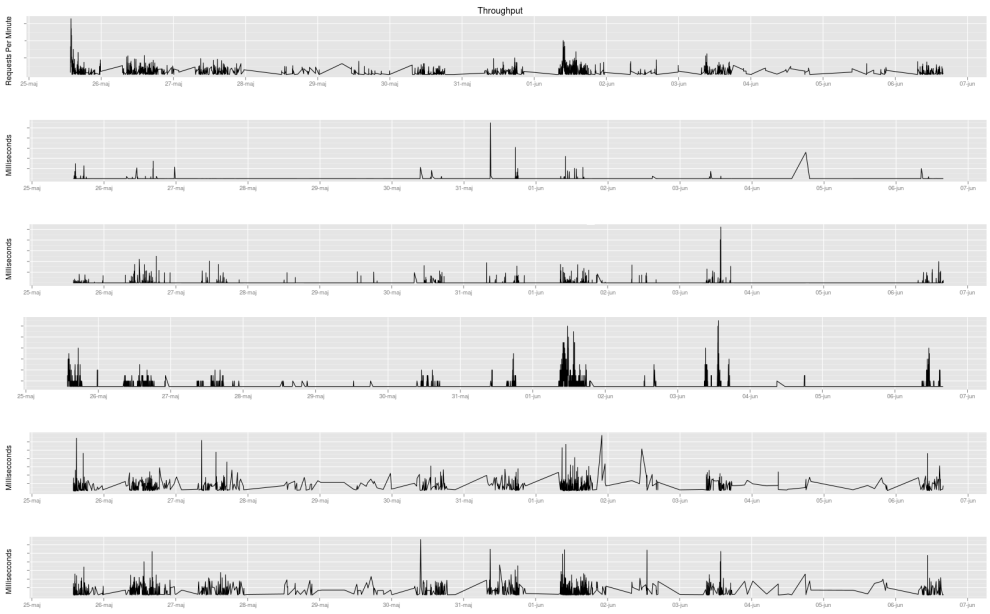
Cherry pick ideas from Basho Bench

- Creates a histogram of timings on the fly, reducing the number of data points
- Dumps to disk every N seconds
- Allows statistical tools to work on already aggregated data
- Near real-time, from event to stats in N+5 seconds

Homegrown stats

- Measures latency from the edges of our system (excludes HTTP handling)
- And at interesting points inside the system
- Statistical analysis using R
- Correlate with HAproxy data
- Produces graphs and data specific to our application

Homegrown stats



wooga
world of gaming

Recap

Measure:

- From an external point of view (HAproxy)
- At the edge of the system (excluding HTTP handling)
- Internals in the single process (eprof)

wooga
world of gaming

Recap

Analyze:

- Aggregated measures
- Statistical properties of measures
  - standard deviation
  - distribution
  - trends

Thanks!

http://www.wooga.com/jobs

knut.nesheim@wooga.com @knutin

paolo.negri@wooga.com @hungryblank