

SCALING ERLANG WEB APPLICATIONS

100 TO 100K USERS AT ONE WEB SERVER

Fernando Benavides (*@elbrujothalcon*)

Inaka Labs

March 29, 2012



HELLO WORLD!

- I'm a developer since I was 10
- I worked with Visual Basic, C#, .NET, Javascript ...
- I switched to functional programming in 2008
- I wrote my thesis project in Haskell
- I'm an Erlang developer since then



HELLO WORLD!

- I'm a developer since I was 10
- I worked with Visual Basic, C#, .NET, Javascript . . .
- I switched to functional programming in 2008
- I wrote my thesis project in Haskell
- I'm an Erlang developer since then



HELLO WORLD!

- I'm a developer since I was 10
- I worked with Visual Basic, C#, .NET, Javascript . . .
- I switched to functional programming in 2008
- I wrote my thesis project in Haskell
- I'm an Erlang developer since then



HELLO WORLD!

- I'm a developer since I was 10
- I worked with Visual Basic, C#, .NET, Javascript . . .
- I switched to functional programming in 2008
- I wrote my thesis project in Haskell
- I'm an Erlang developer since then



HELLO WORLD!

- I'm a developer since I was 10
- I worked with Visual Basic, C#, .NET, Javascript ...
- I switched to functional programming in 2008
- I wrote my thesis project in Haskell
- I'm an Erlang developer since then



INAKA

- We're based in Argentina
- We do Erlang consulting
- We often build end-to-end applications with
 - Erlang backends
 - iOS clients
 - Ruby admin tools



INTRODUCTION

My talk is on the scalability of a *web* project that has an *HTTP API* and a component that keeps clients *connected* to the server for *long periods* of time.

It's a design pattern seen in many places:

- Chat Rooms
- Social Sites
- Sport Sites



INTRODUCTION

My talk is on the scalability of a *web* project that has an *HTTP API* and a component that keeps clients *connected* to the server for *long periods* of time.

It's a design pattern seen in many places:

- Chat Rooms
- Social Sites
- Sport Sites



INTRODUCTION

My talk is on the scalability of a *web* project that has an *HTTP API* and a component that keeps clients *connected* to the server for *long periods* of time.

It's a design pattern seen in many places:

- Chat Rooms
- Social Sites
- Sport Sites



INTRODUCTION

My talk is on the scalability of a *web* project that has an *HTTP API* and a component that keeps clients *connected* to the server for *long periods* of time.

It's a design pattern seen in many places:

- Chat Rooms
- Social Sites
- Sport Sites



SCOPE

We will improve the way we use

- OTP behaviours
- TCP and HTTP connections
- Underlying system configurations

*We will **not** deal with*

- Multiple machines/nodes
- Database choices and/or implementations



SCOPE

We will improve the way we use

- OTP behaviours
- TCP and HTTP connections
- Underlying system configurations

*We will **not** deal with*

- Multiple machines/nodes
- Database choices and/or implementations



MATCH STREAM

GENERAL IDEA

A soccer match is played at some stadium



MATCH STREAM

GENERAL IDEA

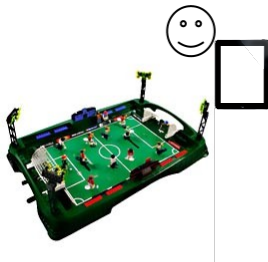
Soccer fans are connected to the internet in their offices



MATCH STREAM

GENERAL IDEA

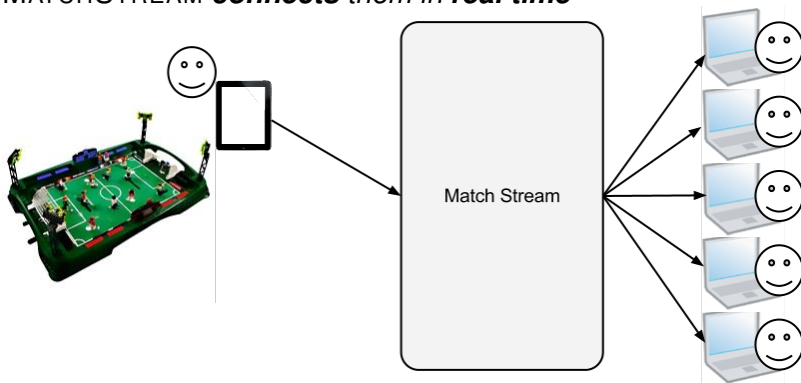
A reporter is at the stadium with his device



MATCH STREAM

GENERAL IDEA

MATCHSTREAM ***connects*** them in ***real time***



MATCH STREAM

REQUIREMENTS

- Many concurrent users connecting at the same time
- Two-hour-long bursts of connections followed by long periods of inactivity
- Real-time updates



MATCH STREAM

REQUIREMENTS

- Many concurrent users connecting at the same time
- Two-hour-long bursts of connections followed by long periods of inactivity
- Real-time updates



MATCH STREAM

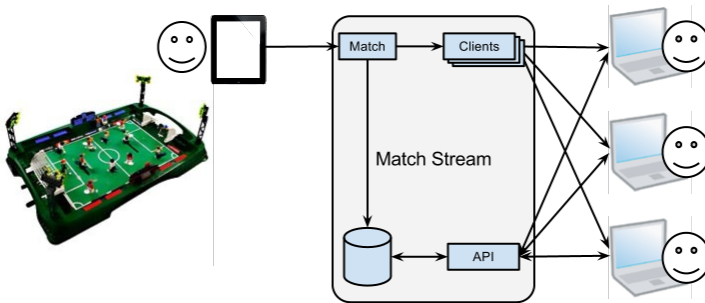
REQUIREMENTS

- Many concurrent users connecting at the same time
- Two-hour-long bursts of connections followed by long periods of inactivity
- Real-time updates



MATCH STREAM

GENERAL DESIGN



MATCH STREAM

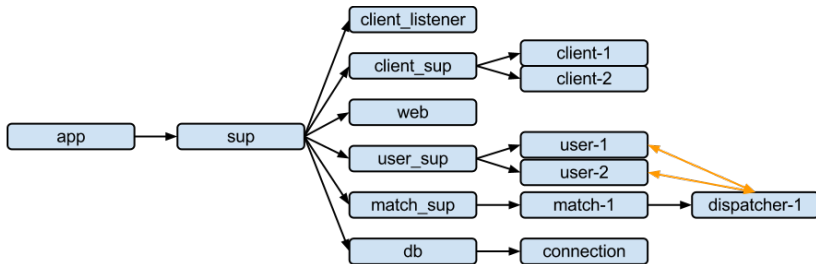
SAMPLE

```
> telnet <server> <port> | 2011-09-13 13:48:51: goal:
... | player: Luna (7)
Welcome to Match Stream. | team: <<"tig">>
... |
V:2:elbrujoalcon | 2011-09-13 13:49:03: penalty:
2011-09-13 13:48:48: status: | player: Martinez (6)
    home: <<"elp">> | team: <<"tig">>
    home_players: |
        Albil (25) | 2011-09-13 13:49:04: card:
        ... | player: Albil (25)
    home_score: 0 | card: red
    visit: <<"tig">> | team: <<"elp">>
    visit_players: |
        .. | 2011-09-13 13:49:05: substitution:
    visit_score: 0 | player_out: Fernandez (18)
    period: first | team: <<"elp">>
                | player_in: Silva (21)
```

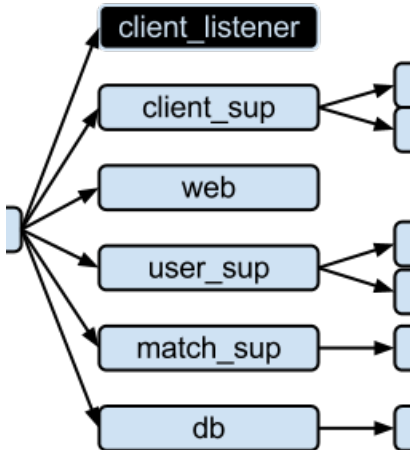


MATCH STREAM

ARCHITECTURE

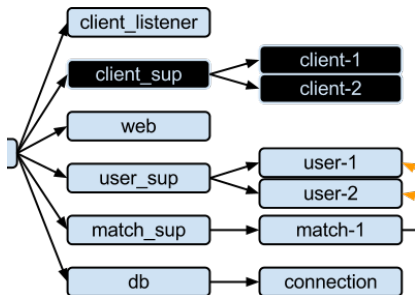


COMPONENTS



`CLIENT_LISTENER` `gen_server`.
Listens on a TCP
port to receive
client connections

COMPONENTS

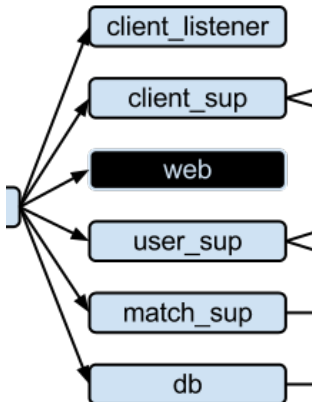


CLIENT_SUP supervisor.
 Supervises
 connection
 processes

CLIENT `gen_fsm`.
 Handles a TCP
 connection



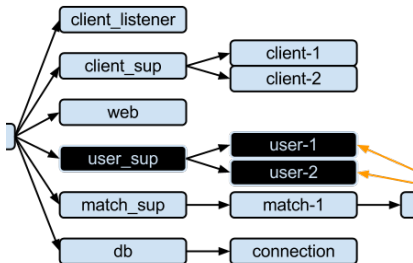
COMPONENTS



WEB mochiweb server.
Listens for HTTP
API calls



COMPONENTS

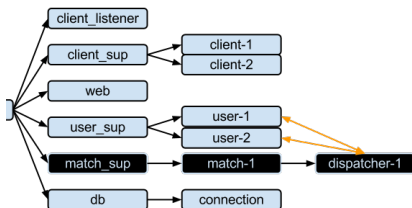


`USER_SUP` supervisor.
 Supervises user processes

`USER` gen_server.
 Subscribes to match dispatchers and sends events to clients



COMPONENTS



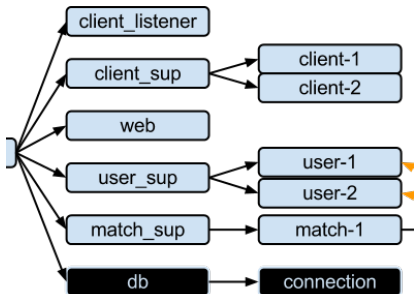
MATCH_SUP supervisor.
 Supervises match processes

MATCH gen_server.
 Listens to match events, stores them

DISPATCHER gen_event dispatcher.
 Delivers match events



COMPONENTS



DB `gen_server`.
 Processes
 database
 operations

CONNECTION `erldis client`.
 Handles the
 connection to the
 database



LESSON LEARNED

*Simply using Erlang to build your system is **not enough** to ensure **scalability***



MEASURES

- N** *Connections*. Number of connections the server can handle
- C** *Concurrency*. Number of connections starting at a time
- ART** *Average Response Time*. How much time it takes for the server to send an event



TOOLS

TEST CLIENT

We create our own test client for TCP connections

APACHEBENCH

To test API calls

ENTOP

We use it to check server processes



STAGE 0

ESTABLISHING A BASELINE

GOALS

- Find how much the system can handle

STEPS

- Create automated testers
- Install and start the system on a *clean* machine
- Run the tests on the server's local network
- Test repeatedly adjusting our parameters to maximize them
- Have a human using the system himself



STAGE 0

ESTABLISHING A BASELINE

GOALS

- Find how much the system can handle

STEPS

- Create automated testers
- Install and start the system on a *clean* machine
- Run the tests on the server's local network
- Test repeatedly adjusting our parameters to maximize them
- Have a human using the system himself



STAGE 1

RESULTS



N **1000**
C **4**
ART **26s**



STAGE 1

TUNE THE OS AND THE VM

GOALS

- Improve the underlying Operating System
- Improve the Erlang VM Configuration

SETTINGS TO TUNE UP

- Open files limit
- TCP connections limit
- TCP backlog size
- TCP memory allocation
- Number of Erlang processes



STAGE 1

TUNE THE OS AND THE VM

GOALS

- Improve the underlying Operating System
- Improve the Erlang VM Configuration

SETTINGS TO TUNE UP

- Open files limit
- TCP connections limit
- TCP backlog size
- TCP memory allocation
- Number of Erlang processes



STAGE 1

RESULTS



N **4000**
C **4**
ART **31s**



STAGE 2.1

CONNECTION TWEAKS

BACKLOG

- It defaults to 5, we should increase it
- Don't forget to TCP tune your HTTP server

The backlog value defines the maximum length that the queue of pending connections may grow to.
(Erlang Docs)



STAGE 2.1

CONNECTION TWEAKS

CLIENT_LISTENER

```
gen_tcp:listen(Port,  
  [binary, {packet, line}, {keepalive, true},  
   {active, false}, {reuseaddr, true},  
   {backlog, 128000}, {send_timeout, 32000},  
   {send_timeout_close, true}]).
```

WEB

```
mochiweb_http:start(  
  [{name, ?MODULE}, {loop, {?MODULE, loop}},  
   {backlog, 128000}, {port, Port}]).
```



STAGE 2.1

CONNECTION TWEAKS

OUTBOUND CONNECTIONS

- e.g., database connections
- Don't use just one of them
- You may have separate connections for different purposes



STAGE 2.1

CONNECTION TWEAKS

```
-define(REDIS_CONNECTIONS, 200).  
-record(state, {redis :: [pid()] }).  
...  
init([]) ->  
...  
Redis =  
    lists:map(  
        fun(_) ->  
            {ok, Conn} = erldis_client:start_link()  
            Conn  
        end, lists:seq(1, ?REDIS_CONNECTIONS) ),  
{ok, #state{redis = Redis}}.
```



STAGE 2.1

CONNECTION TWEAKS

```
handle_call(Request, From, State) ->
  [RedisConn|Redis] = State#state.redis,
  proc_lib:spawn_link(
    fun() ->
      Res = handle_call(Request, RedisConn),
      gen_server:reply(From, Res)
    end),
  {noreply, State#state{redis =
    Redis ++ [RedisConn] }}.
```



STAGE 2.1

CONNECTION TWEAKS

LISTENERS

- You can listen to more than one port
- For unified urls, use *nginx* in front of the server



STAGE 2.1

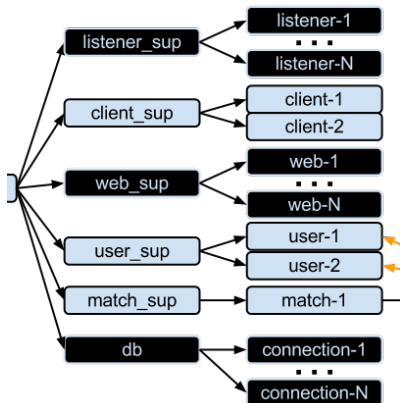
CONNECTION TWEAKS

```
init([]) ->
...
Listeners =
    [{list_to_atom("client-listener-" ++
                    integer_to_list(I)),
      {client_listener, start_link, [I]},
      permanent, brutal_kill, worker,
      [client_listener]}
    || I <- lists:seq(MinPort, MaxPort) ],
{ok, {{one_for_one, 5, 10}, Listeners}}.
```



STAGE 2.1

CONNECTION TWEAKS



LISTENER `gen_server`.
Listens on a TCP port to receive client connections

WEB `mochiweb` server.
Listens for HTTP API calls on a particular port

CONNECTION `erldis` client.
Handles the connection to the database



STAGE 2.1

RESULTS



N **8000**
C **500**
ART **15s**



STAGE 2.2

GEN_EVENT

SUP_HANDLER

- Don't use it
- It exponentially increases the number of 'EXIT' messages sent to the subscribers because it notifies anybody of any process termination
- Monitor the processes instead



STAGE 2.2

GEN_EVENT

```
EvtMgr =  
    match_stream_match:event_manager(MatchId),  
ok =  
    gen_event:add_handler(EvtMgr,  
        {?MODULE, {MatchId,UserId,Client}}, self()),  
MgrRef = erlang:monitor(process, EvtMgr),  
ClientRef = erlang:monitor(process, Client),  
{reply, ok,  
    State#state{matches =  
        [{Client, MatchId, ClientRef, MgrRef }  
        | State#state.matches]}}
```



STAGE 2.2

GEN_EVENT

```
handle_info({ 'DOWN', Ref, _, Client, _ }, State) ->
...
case lists:keytake(Ref, 4, State#state.matches) of
  {value, {Client, _, CRef, Ref}, OtherMatches} ->
    ...
```



STAGE 2.2

GEN_EVENT

LONG DELIVERY QUEUES

- The more subscribers a `gen_event` has, the longer it takes for it to process a notification
- Distribute the work
- Use `gen_event` *repeaters*



STAGE 2.2

GEN_EVENT

```
start_link(Name, Source) ->
    {ok, Pid} = gen_event:start_link(Name),
    ok = gen_event:add_handler(
        Source, {?MODULE, Pid}, Pid),
    {ok, Pid}.

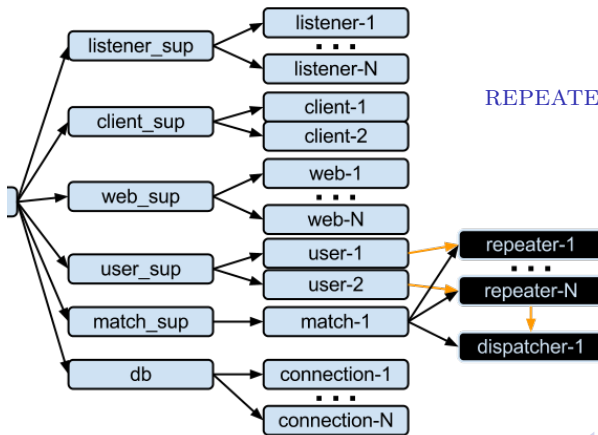
...
init(Repeater) ->
    {ok, #state{mgr = Repeater}}.

...
handle_event(Event, State) ->
    gen_event:notify(State#state.mgr, Event),
    {ok, State}.
```



STAGE 2.2

GEN_EVENT



REPEATER `gen_event` dispatcher. It's subscribed to dispatcher and *repeats* the received events to its subscribers



STAGE 2.2

RESULTS



N **8000**
C **1000**
ART **2s**



STAGE 2.3

GEN_SERVER

QUEUED CALLS

Remember `gen_server:reply/2`



STAGE 2.3

GEN_SERVER

```
handle_call(Request, From, State) ->
  [RedisConn|Redis] = State#state.redis,
  proc_lib:spawn_link(
    fun() ->
      Res = handle_call(Request, RedisConn),
      gen_server:reply(From, Res)
    end),
  {noreply, State#state{redis =
                                Redis ++ [RedisConn]}}.
```



STAGE 2.3

GEN_SERVER

MEMORY FOOTPRINT

Remember `hibernate`

Puts the calling process into a wait state where its memory allocation has been reduced as much as possible, which is useful if the process does not expect to receive any messages in the near future.
(Erlang Docs)



STAGE 2.3

GEN_SERVER

```
handle_cast(Event, State) ->
...
{noreply, State, hibernate}.

...

handle_call(Request, _From, State) ->
...
{reply, Reply, State, hibernate}.
```



STAGE 2.3

GEN_SERVER

LONG STARTUP TIME

- Initialize your gen_servers in a 0 timeout
- Move initialization code to `handle_info`



STAGE 2.3

GEN_SERVER

```
init(UserId) ->
    {ok, #state{user = UserId}, 0}.

...

handle_info(timeout, State) ->
    case match_stream_db:user(State#state.user) of
    ...
```



STAGE 2.3

RESULTS



N **40K**

C **5K**

ART **25ms**



STAGE 2.4

SUPERVISORS

SIMPLE ONE FOR ONE SUPERVISORS

- Sometimes `simple_one_for_one` supervisors get **overburdened** because they have too many children
- Use a supervisor hierarchy



STAGE 2.4

SUPERVISORS

```
init([]) ->
  _ = random:seed(erlang:now()),
  Managers =
    [{list_to_atom("user-manager-" ++
                    integer_to_list(I)),
      {user_mgr, start_link, [I]},
      permanent, brutal_kill, supervisor,
      [user_mgr]}
    || I <- lists:seq(1, ?MANAGERS) ],
  {ok, {{one_for_one, 5, 10}, Managers}}.
```



STAGE 2.4

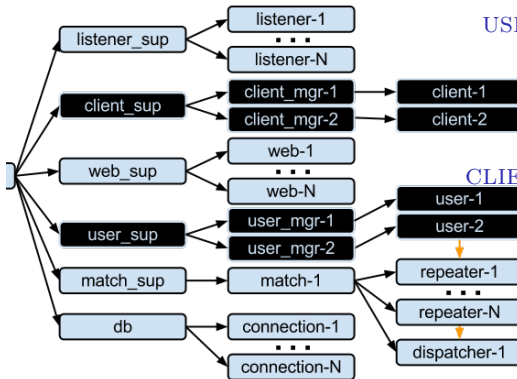
SUPERVISORS

```
start_user(User) ->
  Manager =
    list_to_atom(
      "user-manager-" ++
        integer_to_list(random:uniform(?MANAGERS)),
      supervisor:start_child(Manager, [User]).
```



STAGE 2.4

SUPERVISORS



USER_MGR supervisor.
Supervises a
group of users
processes

CLIENT_MGR supervisor.
Supervises a
group of
connection
processes



STAGE 2.4

RESULTS



N **50K**
C **8K**
ART **25ms**



STAGE 2.5

OTHER PROCESSES

LOGGING

- Use a good logging system
- Choose carefully what to log



STAGE 2.5

OTHER PROCESSES

REGISTRATION

- If everybody agrees on a name, nobody has to find where it is
- You can always register processes **both** locally and globally



STAGE 2.5

OTHER PROCESSES

```
event_manager(MatchId) ->  
    binary_to_atom(<<"event_mgr@", MatchId/binary>>, utf8) .  
  
init(MatchId) ->  
    {ok, EventMgr} =  
        gen_event:start_link(  
            {local, event_manager(MatchId)} ),  
    ...
```



STAGE 2.5

RESULTS



N **50K**
C **8K**
ART **10ms**

STAGE 3

ADDING NODES

GOALS

- Find the best system topology

STEPS

- Prepare the system to run in more than one node
- Decide if nodes should be connected or independent
- Decide if nodes should be on the same machine or not
- Determine which processes should be registered globally



STAGE 3

ADDING NODES

GOALS

- Find the best system topology

STEPS

- Prepare the system to run in more than one node
- Decide if nodes should be connected or independent
- Decide if nodes should be on the same machine or not
- Determine which processes should be registered globally



STAGE 3

ADDING NODES

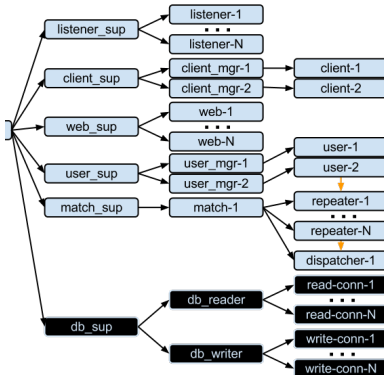
FOR MATCHSTREAM

- We leave the `match` processes registered locally but we connect them using `pg2`
- We split `db` in two:
 - an unique (and therefore) globally registered `db_writer`
 - a `db_reader` per node



STAGE 3

ADDING NODES



DB_READER `gen_server`. One per node.
Processes db read operations

DB_WRITER `gen_server`. One per **system**.
Processes db write operations



STAGE 3

RESULTS



With four nodes in
the same machine:

N **100K**

C **32K**

ART **10ms**



SUMMARY

With started with:

N **1K**
C **4**
ART **26s**

We scale up to:

N **100K**
C **32K**
ART **10ms**

Our improvements:

N **100x**
C **8000x**
ART **2600x**



SUMMARY

With started with:

N **1K**
C **4**
ART **26s**

We scale up to:

N **100K**
C **32K**
ART **10ms**

Our improvements:

N **100x**
C **8000x**
ART **2600x**



SUMMARY

With started with:

N **1K**
C **4**
ART **26s**

We scale up to:

N **100K**
C **32K**
ART **10ms**

Our improvements:

N **100x**
C **8000x**
ART **2600x**



SUMMARY

- This is an **iterative** process
- It proved itself useful in both experimental and real-life systems
- It gets improved with every system we scale



SUMMARY

- This is an **iterative** process
- It proved itself useful in both experimental and real-life systems
- It gets improved with every system we scale



SUMMARY

- This is an **iterative** process
- It proved itself useful in both experimental and real-life systems
- It gets improved with every system we scale



QUESTIONS



RESOURCES

ABOUT ME

- I'm **@elbrujohalcon** at Twitter
- I'm **elbrujohalcon** at Github

ABOUT INAKA

- Check our website (soon :P): <http://inaka.net>
- And our Blog that's on the same site

ABOUT MATCHSTREAM

- The project is on Github:
http://github.com/inaka/match_stream
- This talk, too:
http://github.com/inaka/erlang_factory/brujo



Thanks!

