Scaling Erlang Web Applications

100 to 100K users at one web server

Fernando Benavides (*@elbrujohalcon*)

Inaka Labs

March 26, 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

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 Applications
- 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 Applications
- 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 Applications
- 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 Applications
- Social Sites
- Sport Sites

Scope

We will improve the way we use

- OTP behaviours
- TCP and HTTP connections
- Underlaying 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
- Underlaying 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

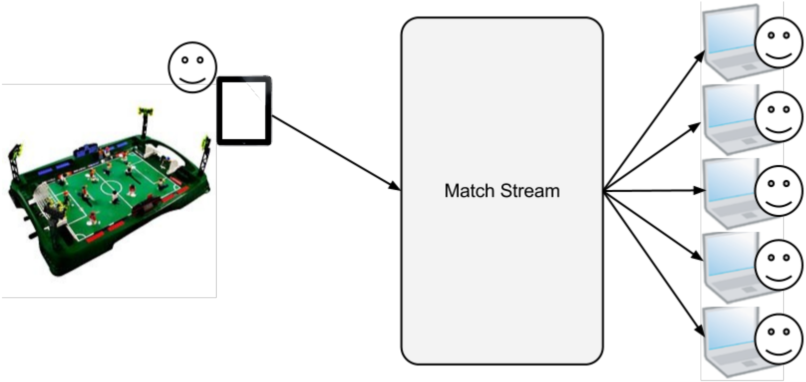
Soccer fans are connected to the internet in their offices

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

System Challenges

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

Erlang seems to be **the right fit for this**

Match Stream
Requirements

System Challenges

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

Erlang seems to be **the right fit for this**

Match Stream
Requirements

System Challenges

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

Erlang seems to be **the right fit for this**

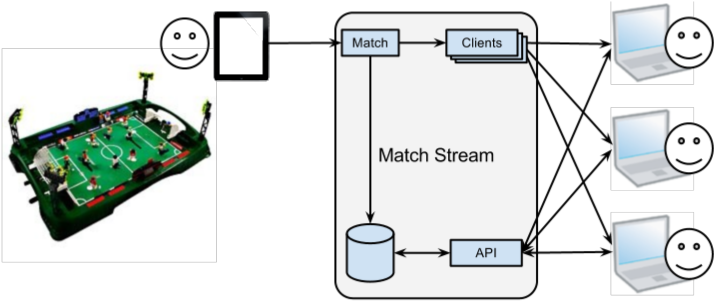
Match Stream
Requirements

System Challenges

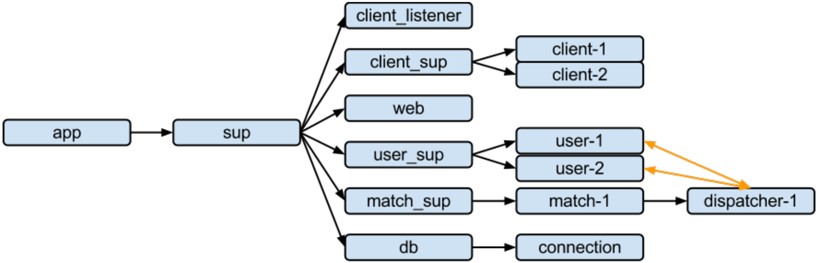
- Many concurrent users connecting at the same time
- Two-hour-long bursts of connections followed by long periods of inactivity
- Real-time updates

Erlang seems to be **the right fit for this**

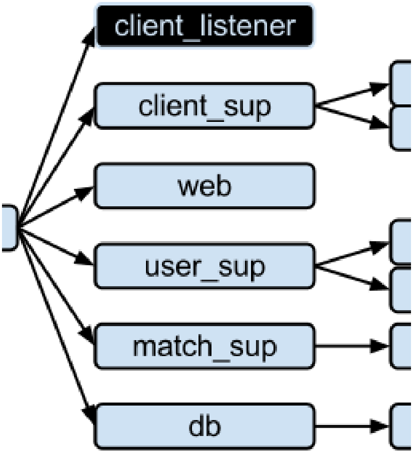
Match Stream
General Design



Match Stream
Architecture

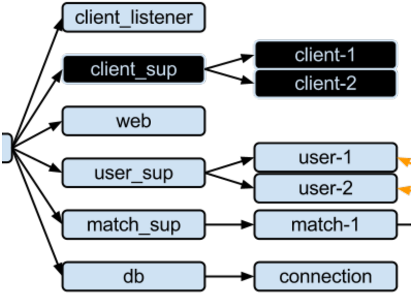


Components



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

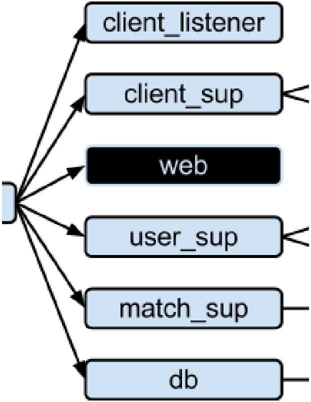
Components



CLIENT_SUP `supervisor`. Supervises connection processes

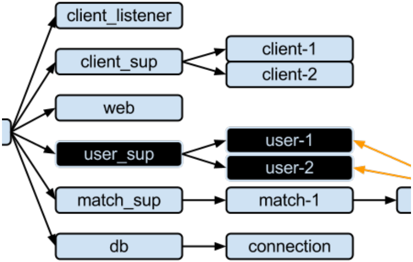
CLIENT `gen_fsm`. Handles a TCP connection

Components



WEB `mochiweb server.`
Listens for HTTP API calls

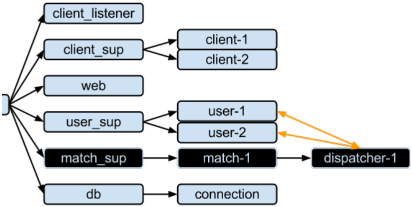
Components



USER_SUP `supervisor`. Supervises user processes

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

Components

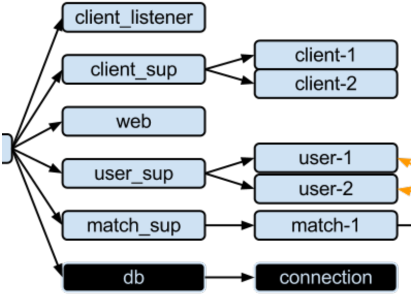


MATCH_SUP `supervisor`. Supervises match processes

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 multiple connections starting at a time
- ART *Average Response Time*. How much does it take 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 see what's going on in the server

Stage 0
Establishing a Baseline

Goals

- Find how much the system can handle

Steps

- Create automated testers
- Start the system on a *clean* machine
- Test repeatedly adjusting the number of connections
- Have a human using the system himself

Stage 0
Establishing a Baseline

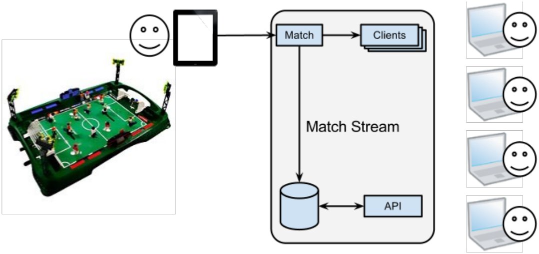
Goals

- Find how much the system can handle

Steps

- Create automated testers
- Start the system on a *clean* machine
- Test repeatedly adjusting the number of connections
- Have a human using the system himself

STAGE 1
RESULTS



N **1000**
C **5**
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
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
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

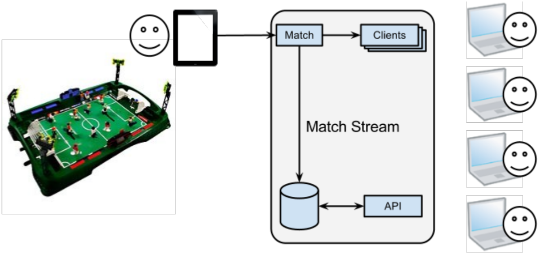
Goals

- 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 **5**
ART **35s**

Stage 2
Improving Match Stream

We can't blame the machine anymore, we need to improve our system

Stage 2.1
Connection Tweaks

Backlog

- Allow more concurrent connections
- Don't forget TCP tuning your HTTP server

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

- For instance, database connections
- Don't use just one of them
- You may have separated connections for different purposes

Stage 2.1
Connection Tweaks

```
-define(REDIS_CONNECTIONS, 200).
-record(state, {redis :: [pid()]}).
...
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

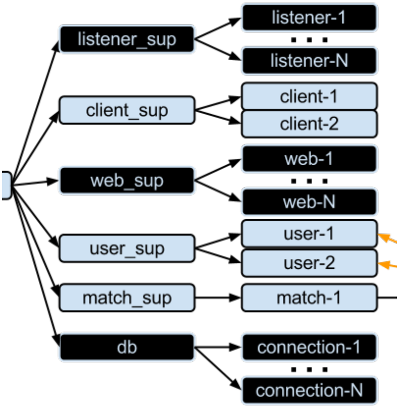
Listeners

- 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

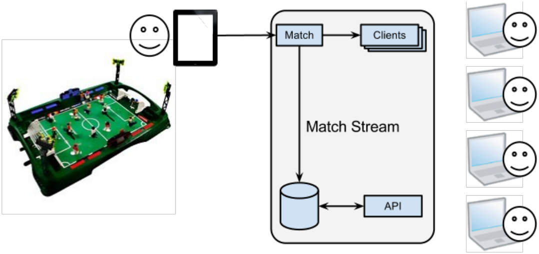


LISTENER `gen_server`. Listens on a TCP port to receive client connections

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
- 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, MatchRef}
    | 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

- Distribute the work
- Use *repeaters*

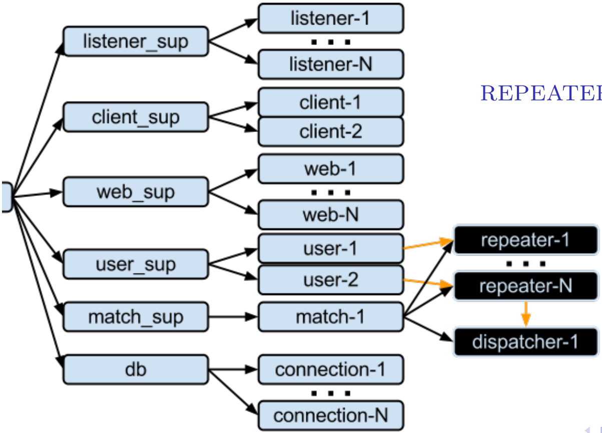
Stage 2.2

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) ->
  Ref = erlang:monitor(process, Repeater),
  {ok, #state{mgr = Repeater, ref = Ref}}.
  ...
handle_event(Event, State) ->
  gen_event:notify(State#state.mgr, Event),
  {ok, State}.
```

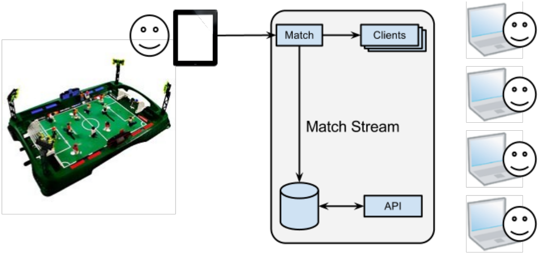
Stage 2.2

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 **500**
ART **8s**

Stage 2.3

GEN_SERVER

Call Timeouts

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

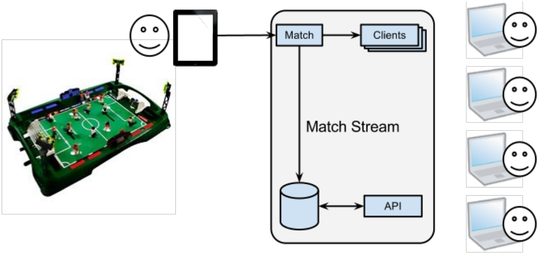
GEN_SERVER

```
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 **32K**
C **2000**
ART **1s**

Stage 2.4
Supervisors

Simple One for Ones

- 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}}.
```

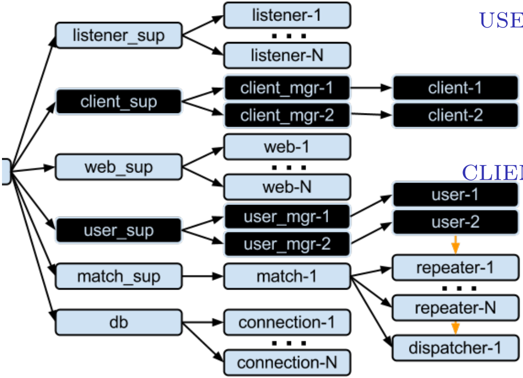
Stage 2.4

Supervisors

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

Stage 2.4

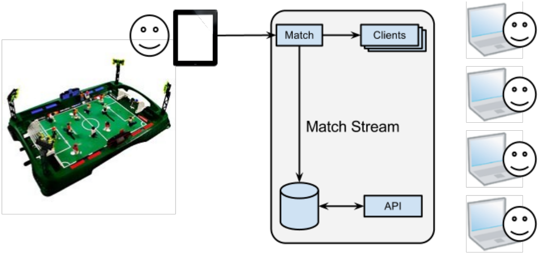
Supervisors



USER_MGR `supervisor`. Supervises a group of users processes

CLIENT_MGR `supervisor`. Supervises a group of connection processes

Stage 2.4
Results



N **64K**
C **4000**
ART **25ms**

Stage 2.5
Other Processes

Logging

- Don’t log too much
- Use a good logging system

Registration

- Sometimes it’s better to register processes instead of keeping track of their pids manually
- You can always register processes both locally and globally

Stage 2.5
Other Processes

Logging

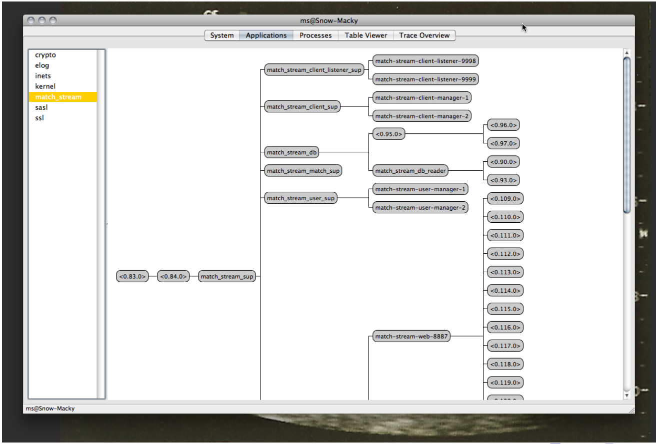
- Don't log too much
- Use a good logging system

Registration

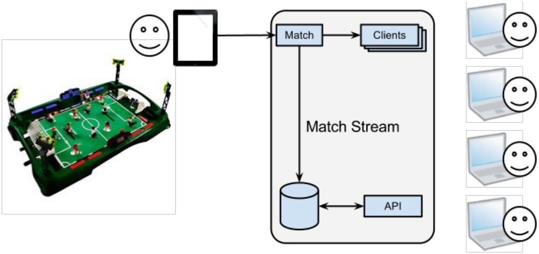
- Sometimes it's better to register processes instead of keeping track of their pids manually
- You can always register processes both locally and globally

STAGE 2.5
OTHER PROCESSES

SYSTEM ARCHITECTURE



Stage 2.5
Results



- **65536** users
- **8192** at a time
- **10ms** ART

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

Stage 3
Adding Nodes

Goals

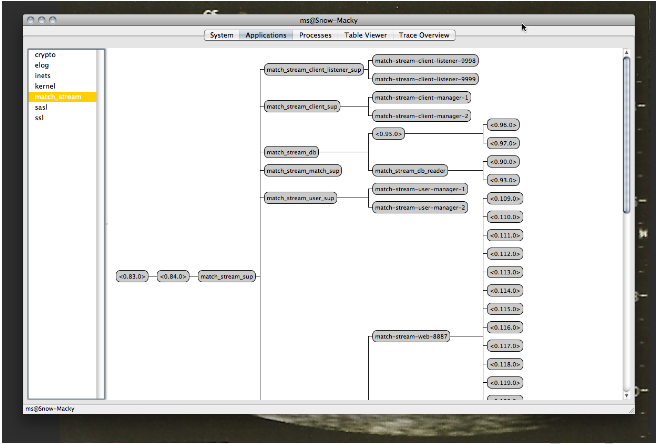
- Find the best system topology

Steps

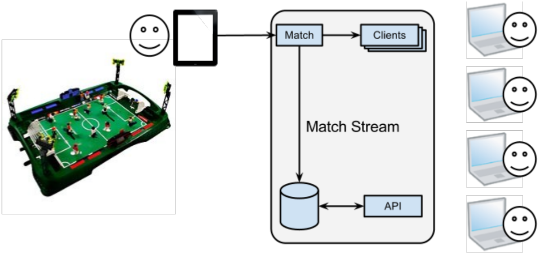
- Prepare the system to run in more than one node
- Decide if nodes should be connected or independent
- Decide if nodes should be on the same machine or not

STAGE 3
ADDING NODES

SYSTEM ARCHITECTURE



Stage 3
Results



- **100K** users
- **32768** at a time
- **10ms** ART

Summary

- This is an iterative process
- It worked awesomely for us in both experimental and real-life systems
- It's no silver bullet
- The list of *Tips and Tricks* grows constantly over time

Summary

- This is an iterative process
- It worked awesomely for us in both experimental and real-life systems
- It's no silver bullet
- The list of *Tips and Tricks* grows constantly over time

Summary

- This is an iterative process
- It worked awesomely for us in both experimental and real-life systems
- It's no silver bullet
- The list of *Tips and Tricks* grows constantly over time

Summary

- This is an iterative process
- It worked awesomely for us in both experimental and real-life systems
- It's no silver bullet
- The list of *Tips and Tricks* grows constantly over time

Scaling Topics

that weren't covered on this presentation

- Managing many nodes
- Choosing databases
- System specific improvements
- Measuring tools

QUESTIONS

Thanks!