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

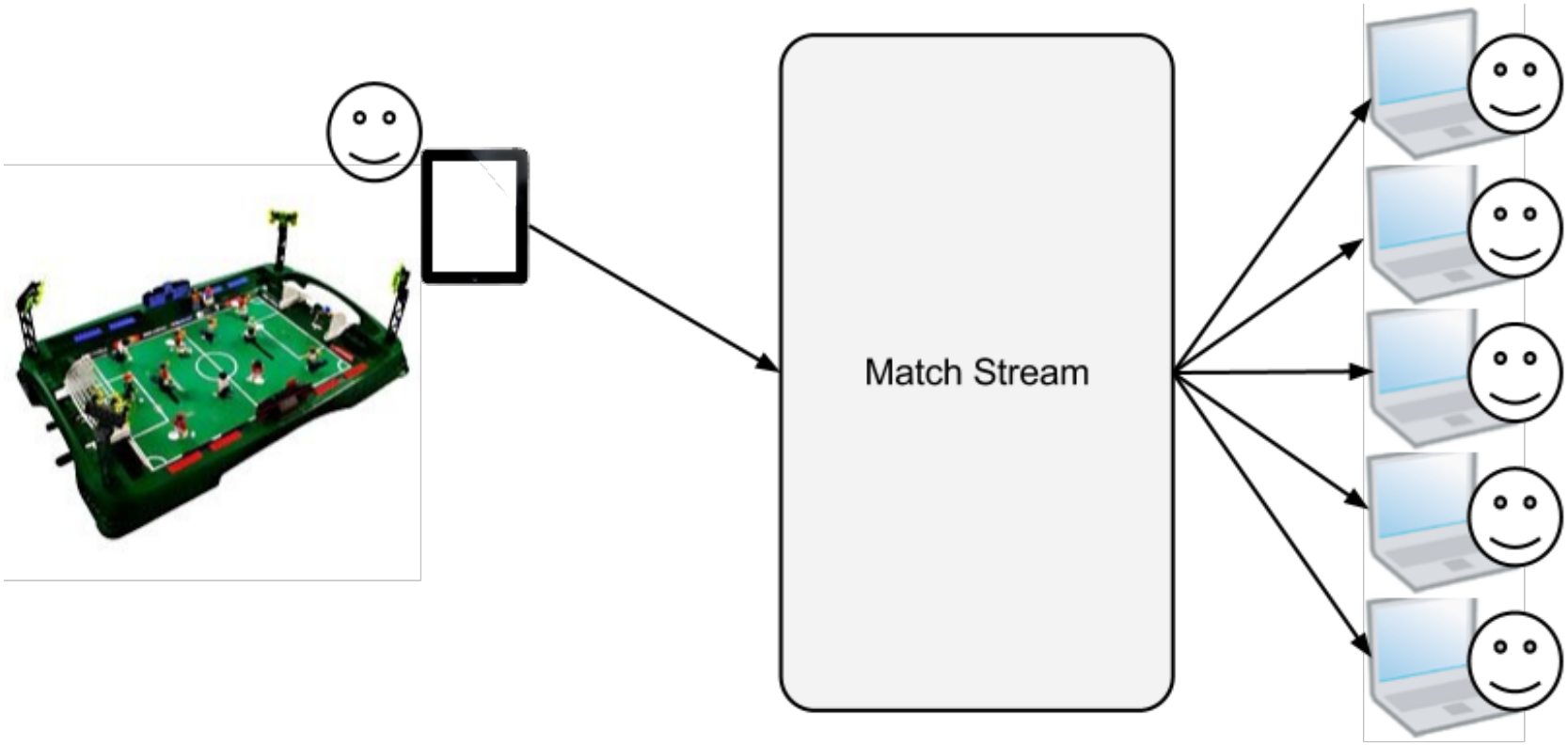
Soccer fans are connected to the internet in their offices

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**

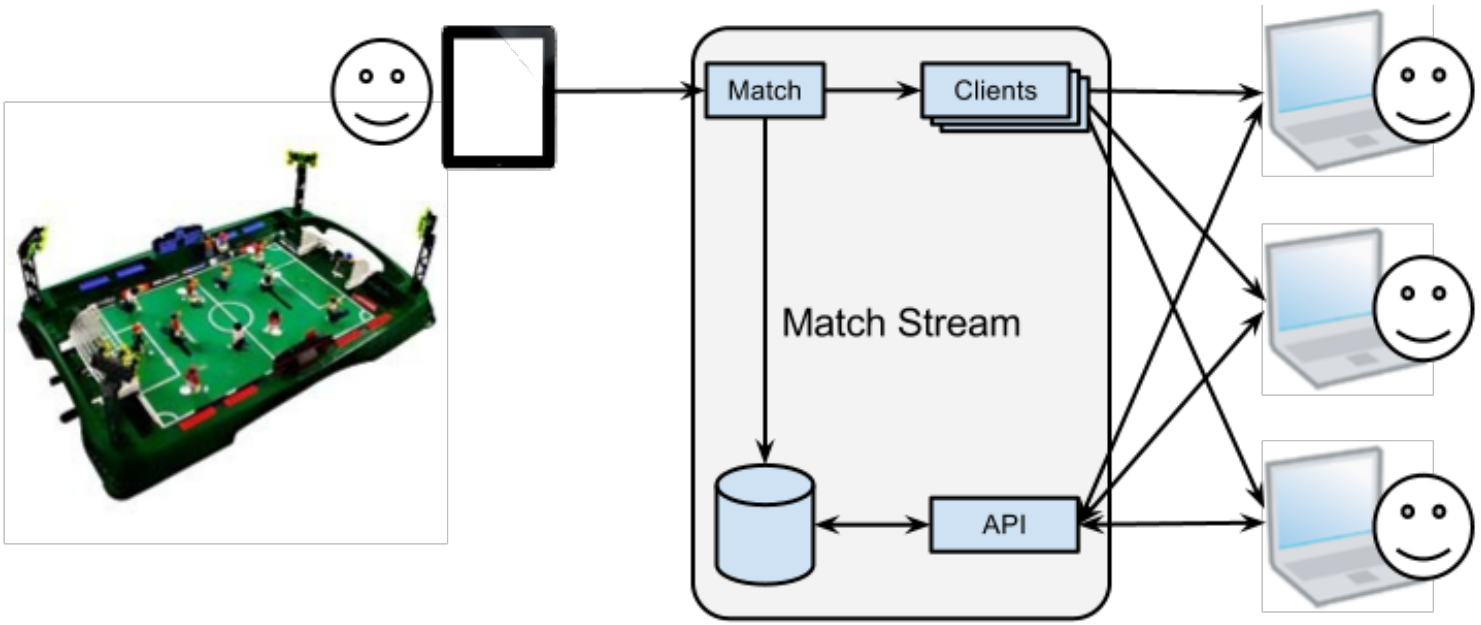
Match Stream
Requirements

System Challenges

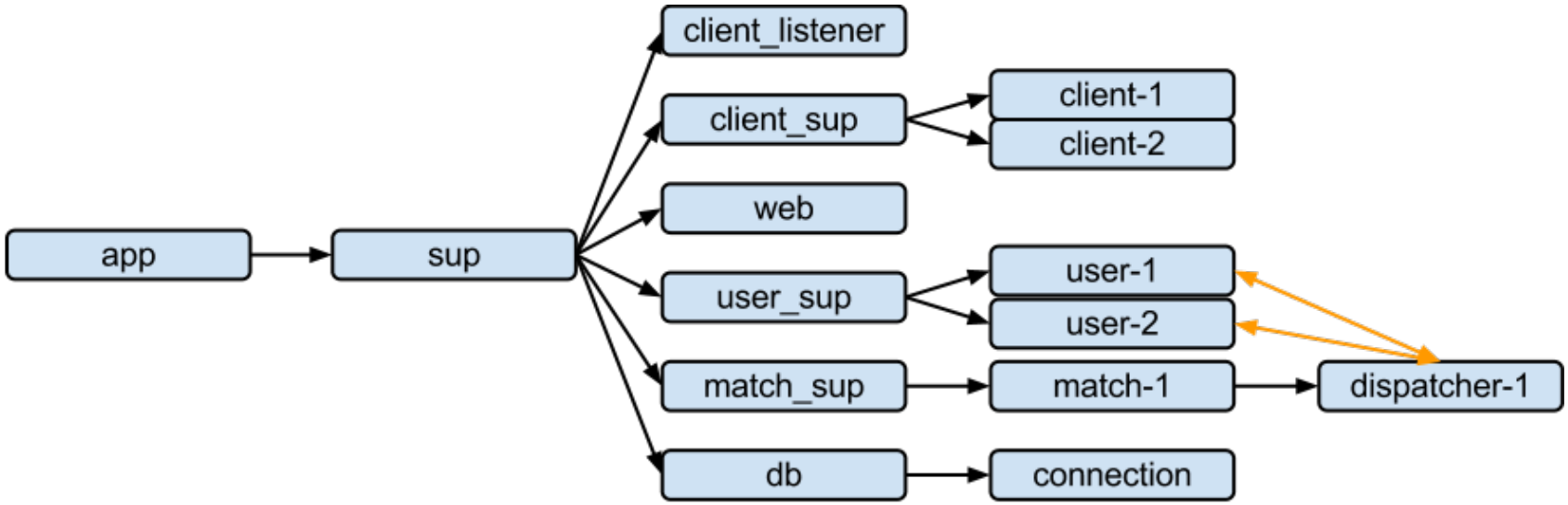
- Many concurrent users connecting at the same time
- Two-hour-long bursts of connections followed by long periods of inactivity
- Real-time updates

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

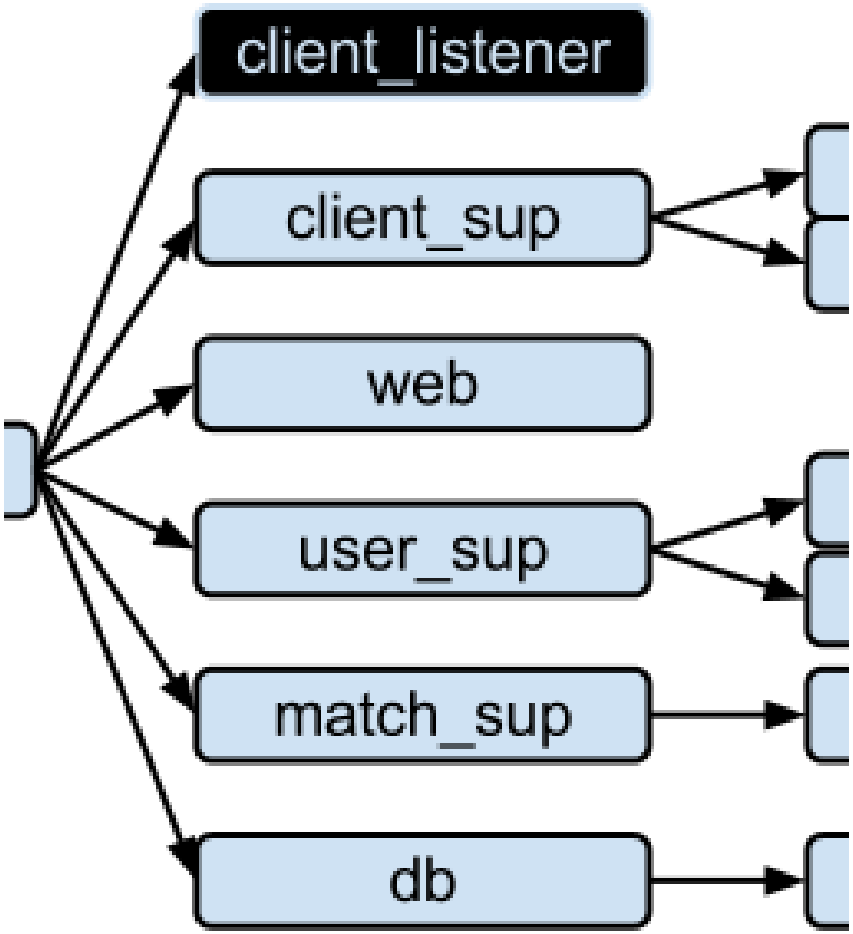
Match Stream
General Design



Match Stream
Architecture



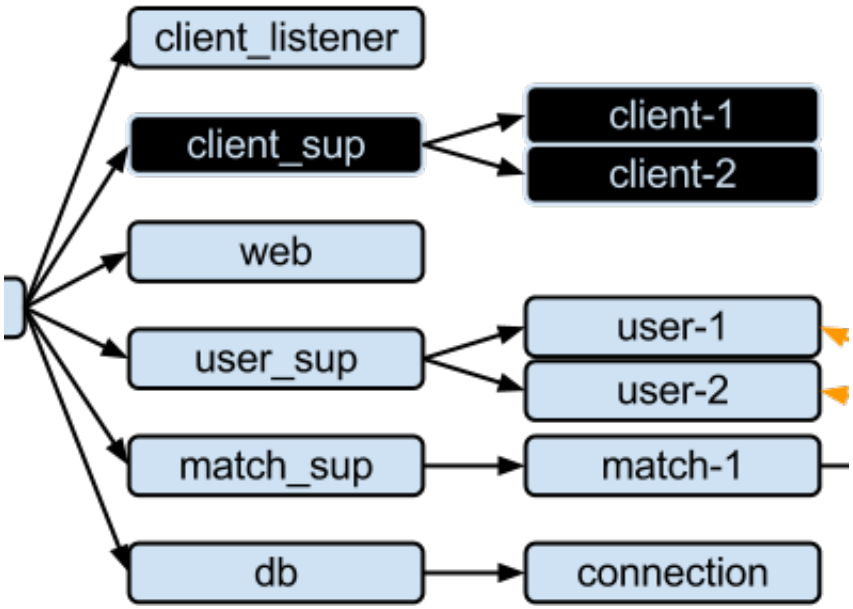
Components



CLIENT_LISTENER `gen_server`.

Listens on a TCP port to receive client connections

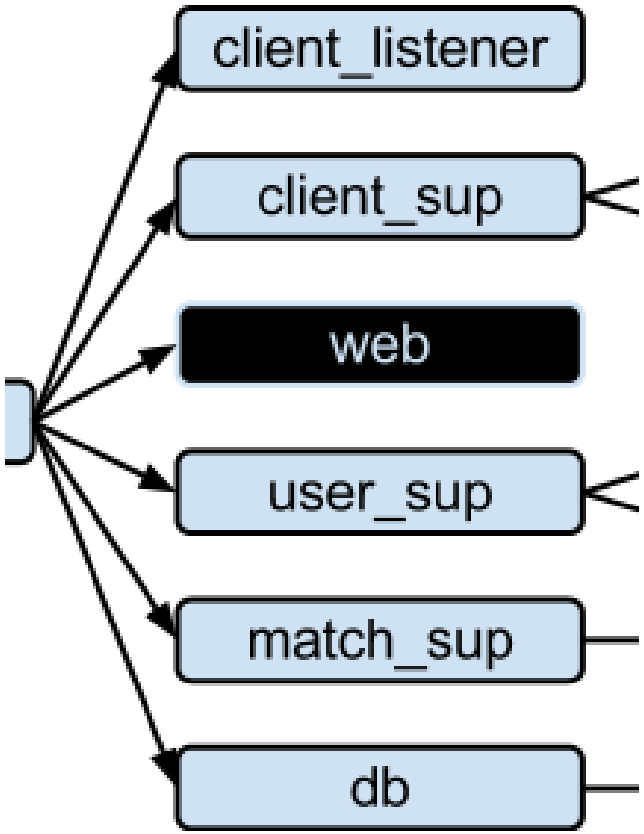
Components



CLIENT_SUP `supervisor.` Supervises connection processes

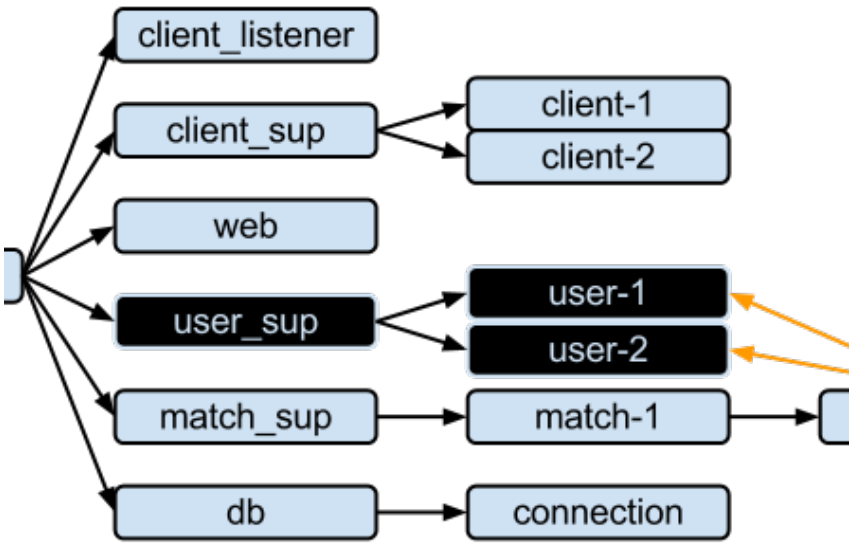
CLIENT `gen_fsm.` Handles a TCP connection

Components



WEB `mochiweb server.`
Listens for HTTP API calls

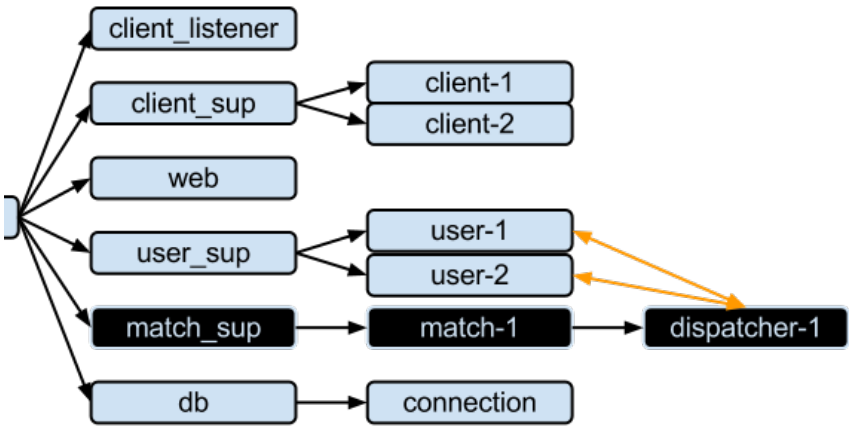
Components



USER_SUP `supervisor`. Supervises user processes

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

Components

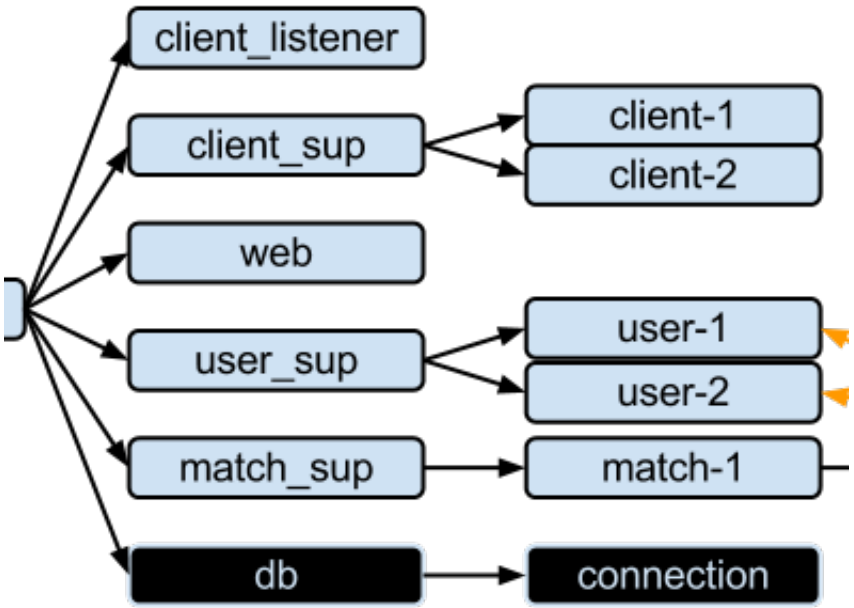


MATCH_SUP `supervisor`. Supervises match processes

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

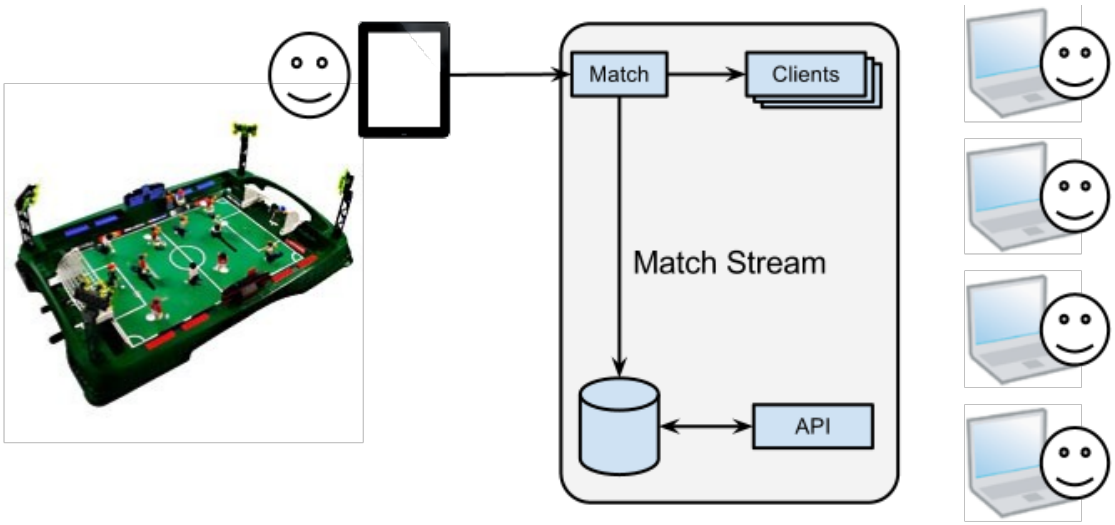
Goals

- 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

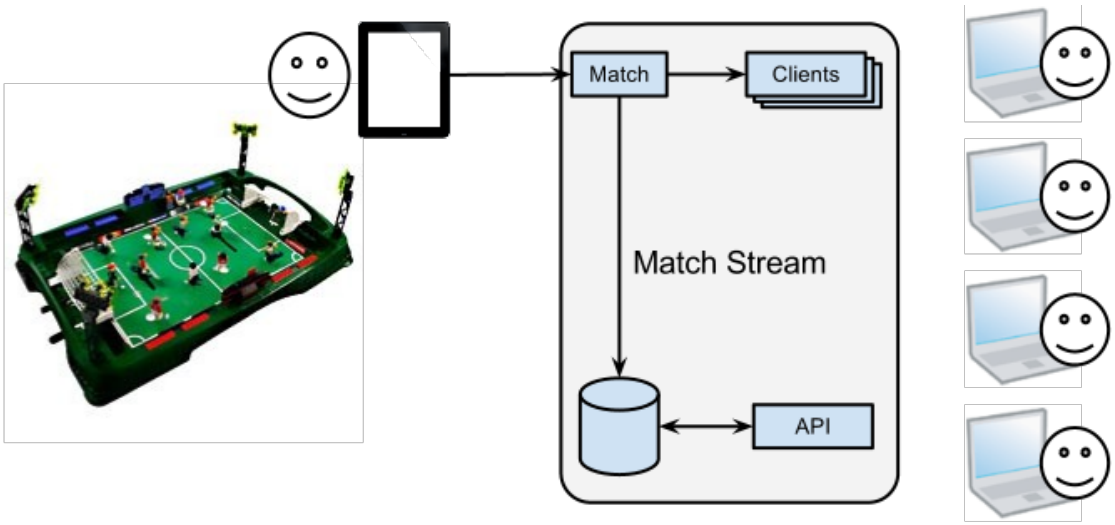
Goals

- 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

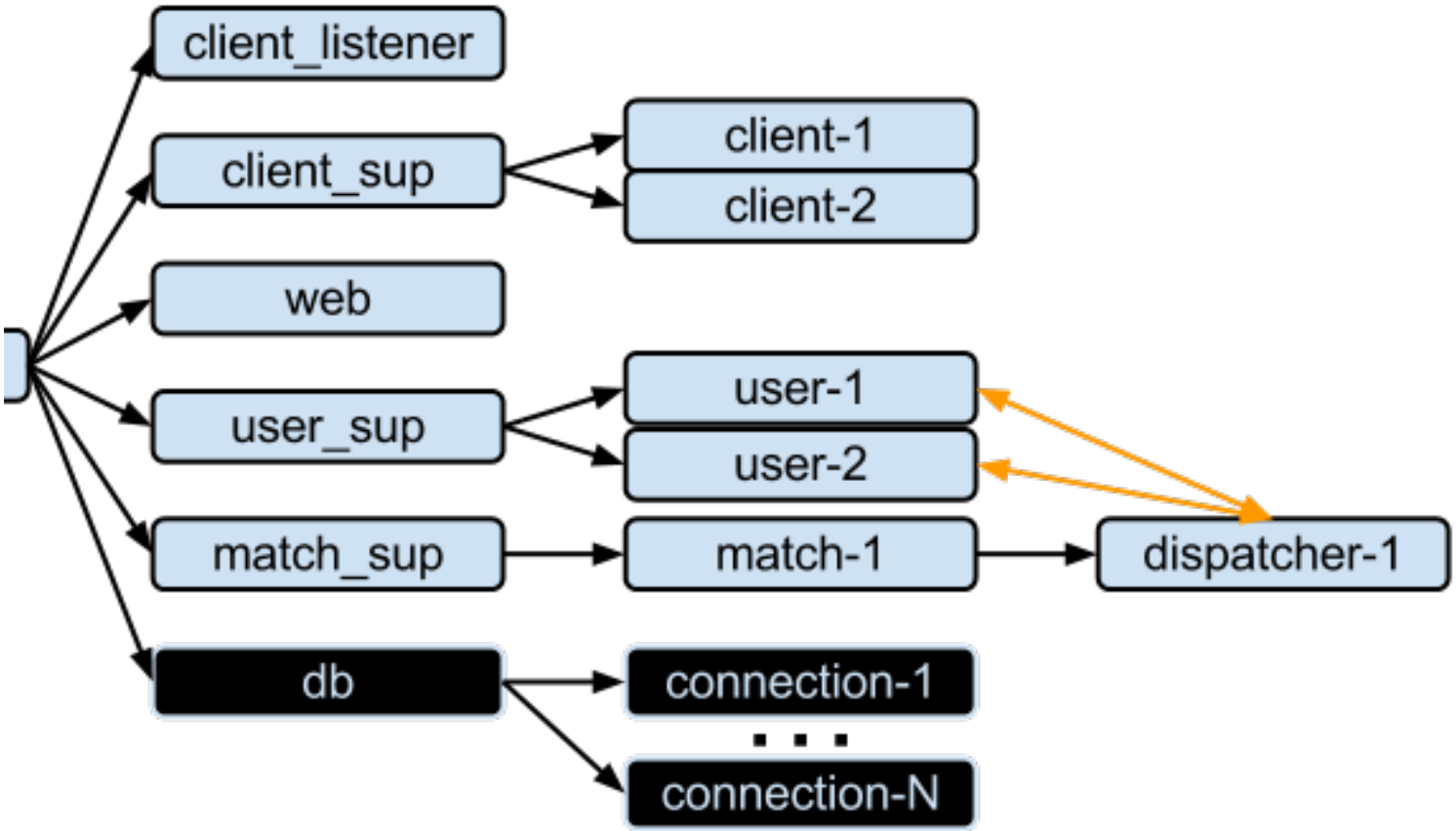
```
init([]) ->
...
  Listeners =
    [{list_to_atom("client-listener-" ++
                   integer_to_list(I)),
      client_listener, start_link, [I],
      permanent, brutal_kill, worker,
      [client_listener]}
     || I <- lists:seq(MinPort, MaxPort)],
  {ok, {{one_for_one, 5, 10}, Listeners}}.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

Stage 2.1
Connection Tweaks

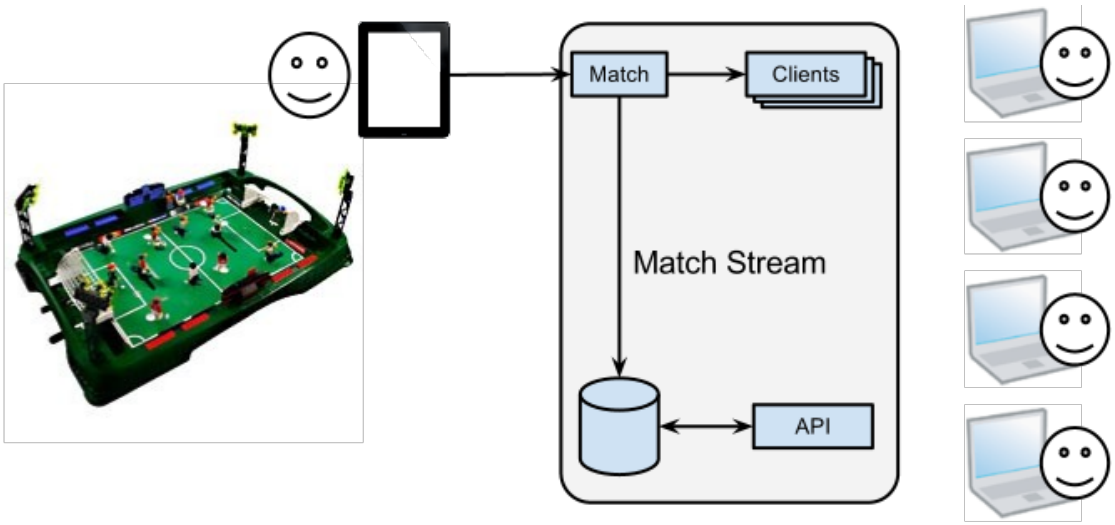
```
handle_call(Request, From, State) ->
  [RedisConn|Redis] = State#state.redis,
  proc_lib:spawn_link(
    fun() ->
      Res = handle_call(Request, RedisConn),
      gen_server:reply(From, Res)
    end),
  {noreply, State#state{redis =
                          Redis ++ [RedisConn]}}.
```

Stage 2.1
Connection Tweaks

System Architecture



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

Long Delivery Queues

- Use *repeaters*

Stage 2.2
gen_event

sup_handler

- Don't use it
- Monitor the processes instead

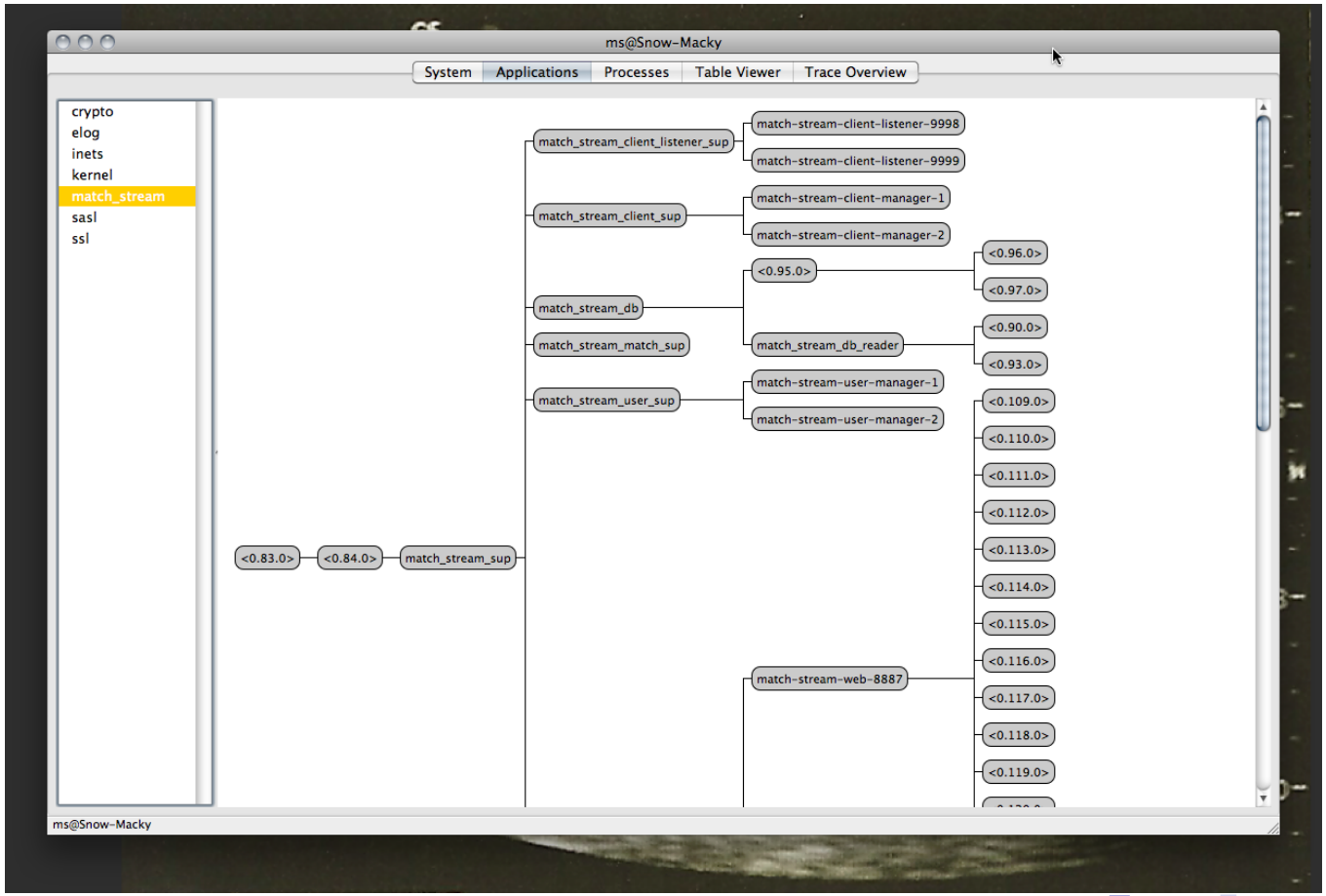
Long Delivery Queues

- Use *repeaters*

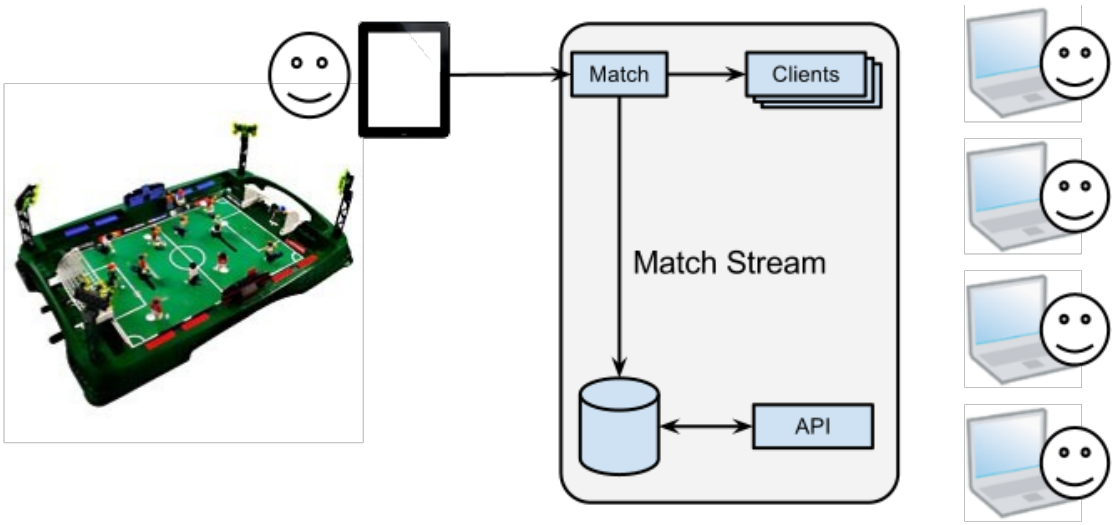
Stage 2.2

gen_event

System Architecture



Stage 2.2
Results



- **8192** users
- **256** at a time
- **8s** ART

Stage 2.3
gen_server

Call Timeouts
Remember `gen_server:reply/2`

Memory Footprint
Remember `hibernate`

Long init/1
Use 0 timeout

Stage 2.3

gen_server

Call Timeouts

Remember `gen_server:reply/2`

Memory Footprint

Remember `hibernate`

Long init/1

Use 0 timeout

Stage 2.3

gen_server

Call Timeouts

Remember `gen_server:reply/2`

Memory Footprint

Remember `hibernate`

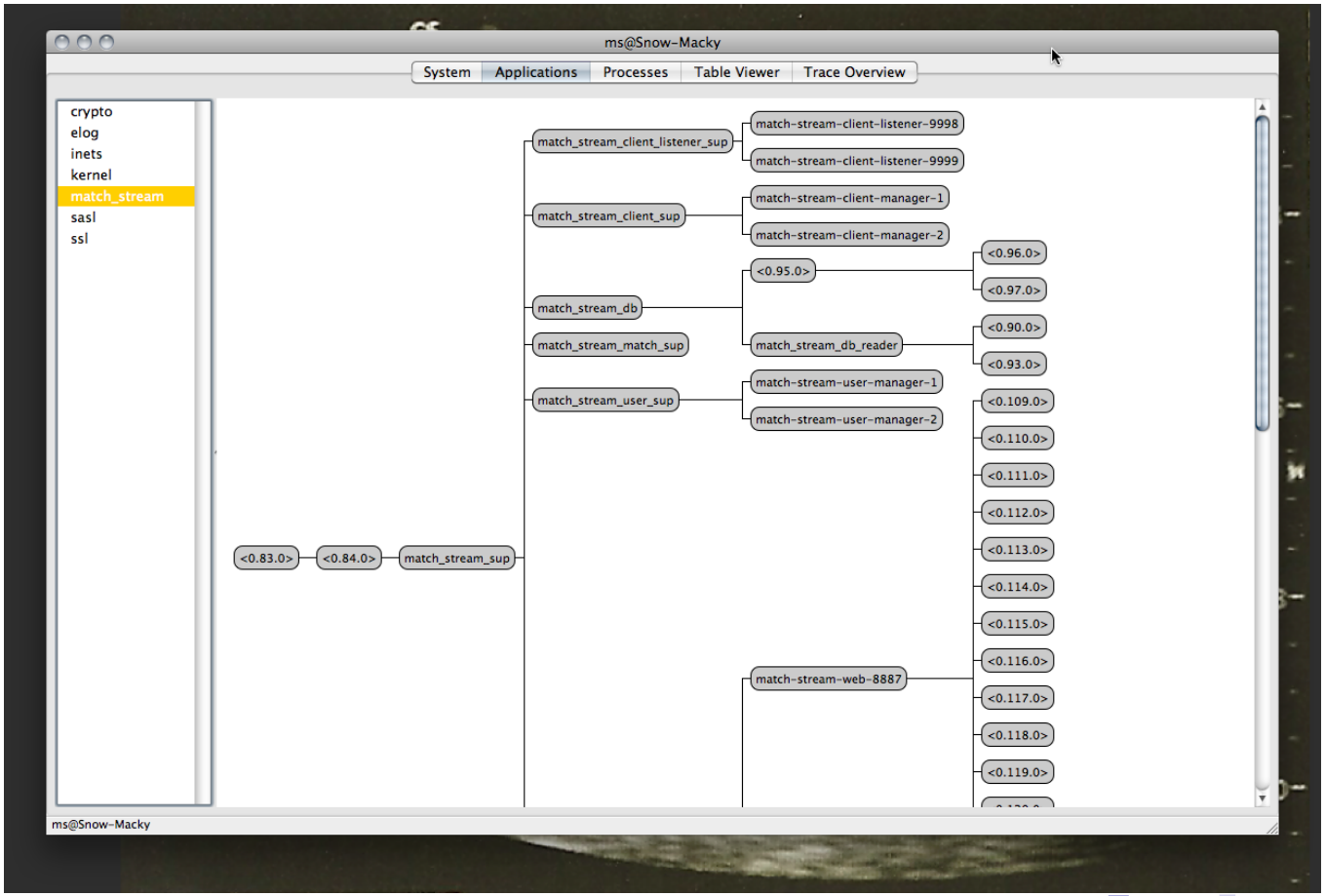
Long init/1

Use 0 timeout

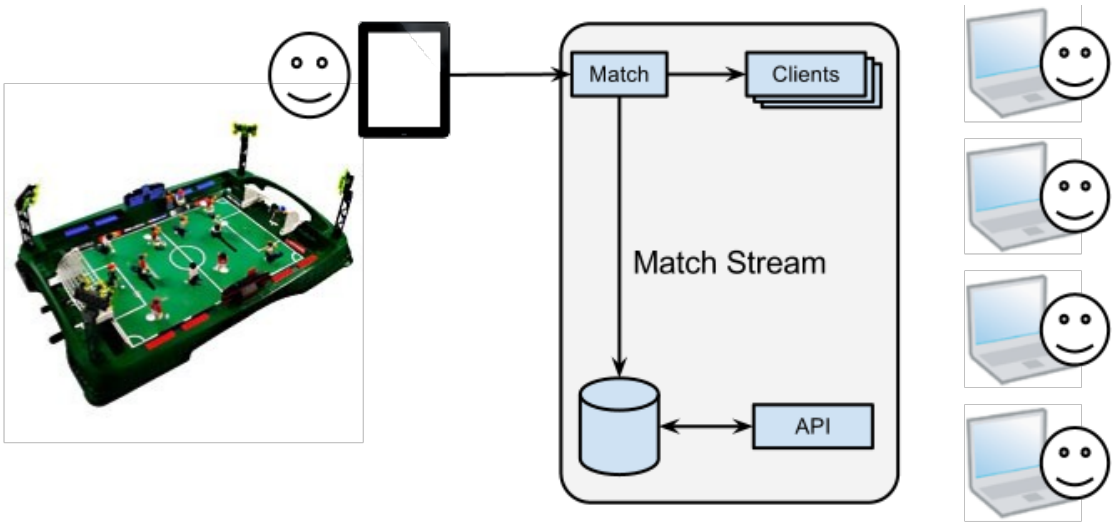
Stage 2.3

gen_server

System Architecture



Stage 2.3
Results



- **32768** users
- **1024** at a time
- **1s** ART

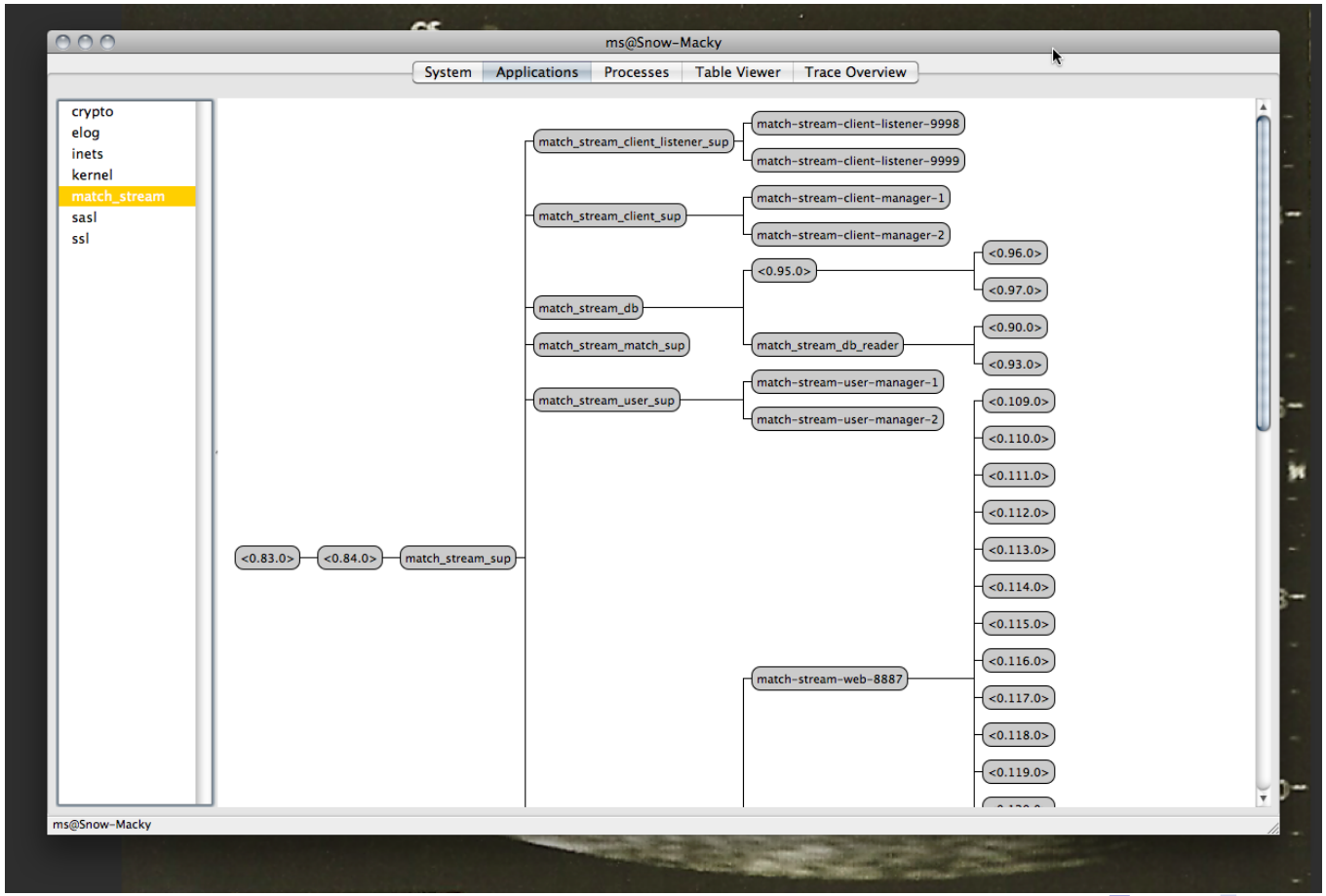
Stage 2.4
supervisors

- Sometimes `simple_one_for_one` supervisors get overburdened because they have too many children
- Try a supervisor hierarchy with several managers below the main supervisor
- Turn `supervisor:start_child/2` calls into something like

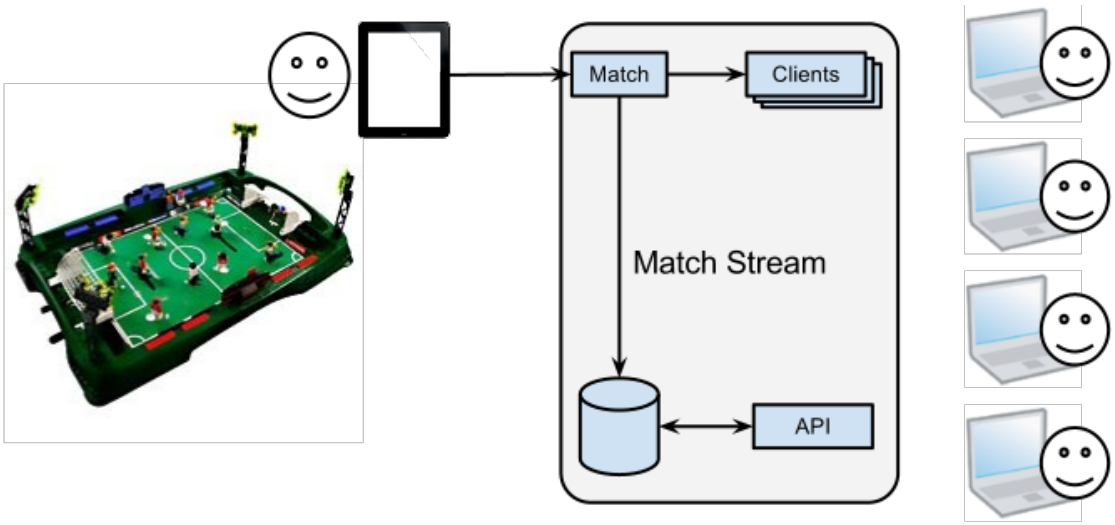
```
supervisor:start_child(
  list_to_atom("module-name_" ++
                                integer_to_list(rando
```

Stage 2.4
supervisors

System Architecture



Stage 2.4
Results



- **65536** users
- **2048** at a time
- **1s** ART

Stage 2.5
Other Processes

Timers

- Don't use the `timer` module
- Use `erlang:send_after`

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

Timers

- Don't use the `timer` module
- Use `erlang:send_after`

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

Timers

- Don't use the `timer` module
- Use `erlang:send_after`

Logging

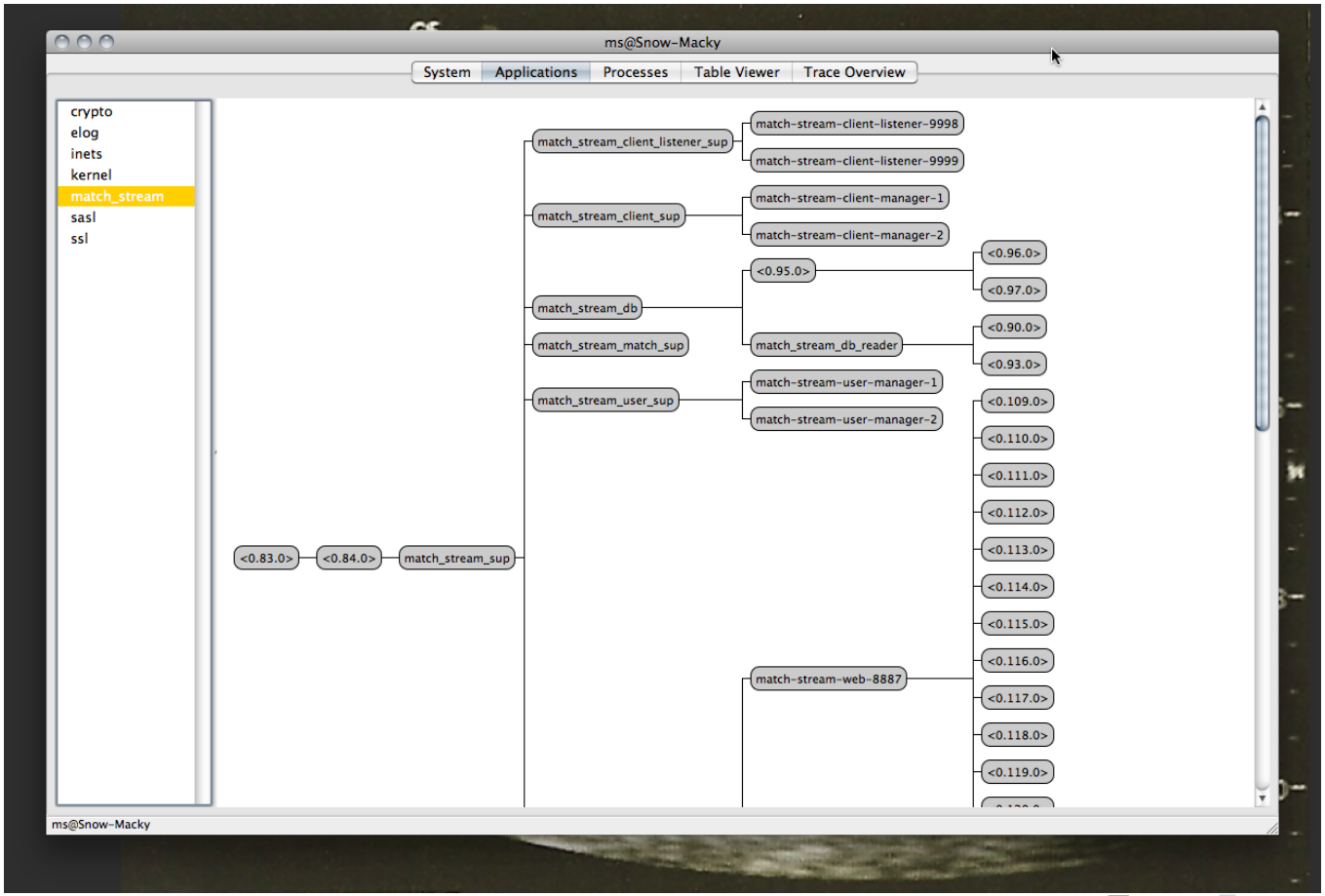
- Don't log too much
- Use a good logging system

Registration

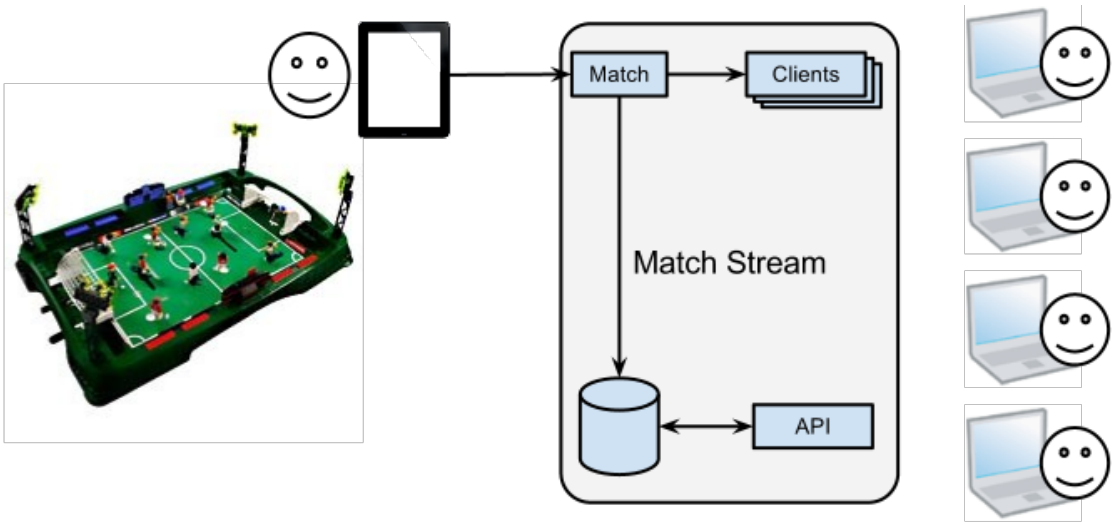
- Sometimes it's better to register processes instead of keeping track of their pids manually
- You can always register processes both locally and globally

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

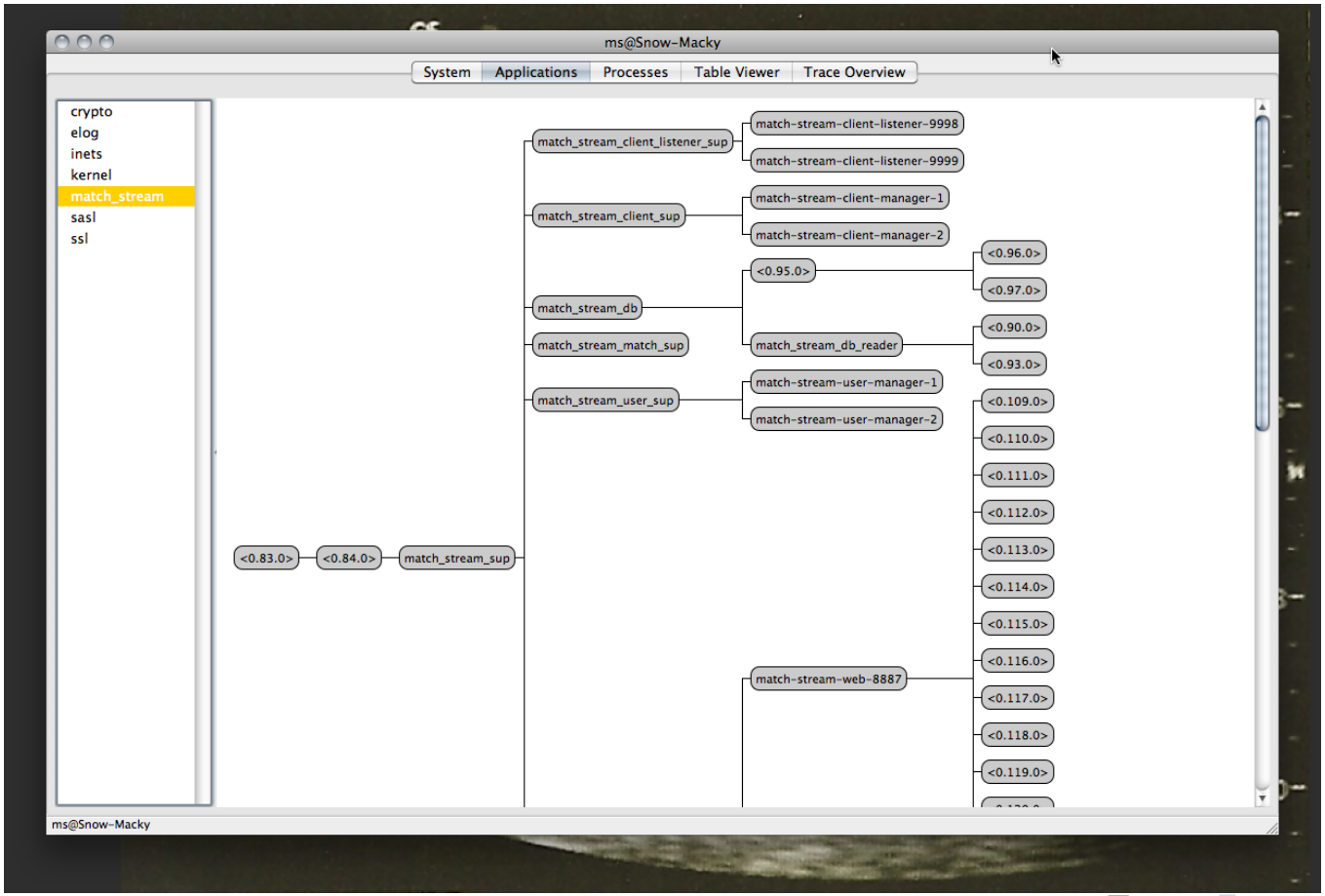
- Find the best system topology

Steps

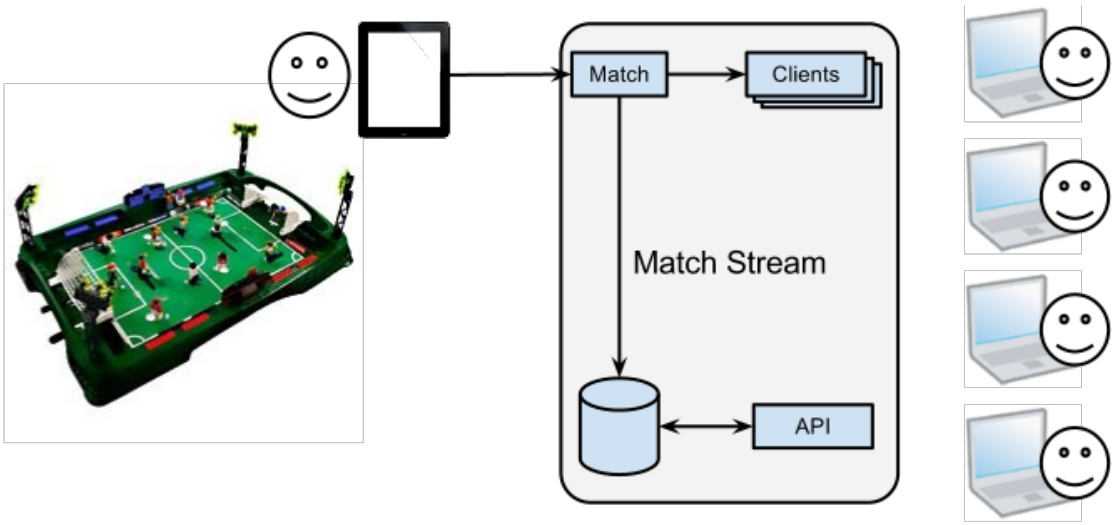
- Prepare the system to run in more than one node
- Decide if nodes should be connected or independent
- Decide if nodes should be on the same machine or not

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!