Scaling Erlang Web Applications

100 to 100K users at one web server

Fernando Benavides (*@elbrujohalcon*)

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

TODO: Chad's speach for this

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
- 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

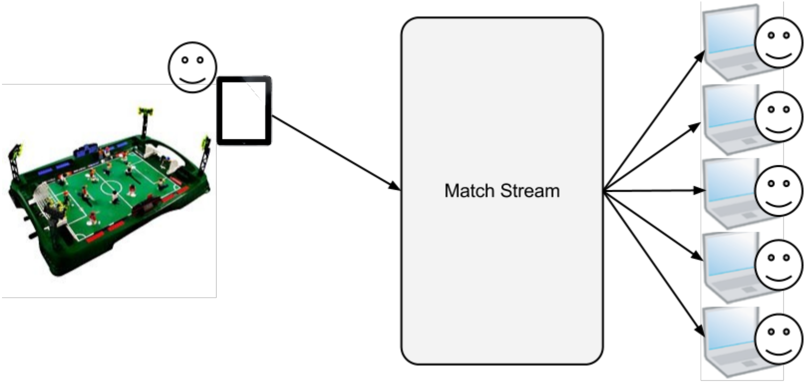
Match Stream

General Idea

A reporter is at the stadium with his device

MATCH STREAM
GENERAL IDEA

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



Match Stream
Requirements

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

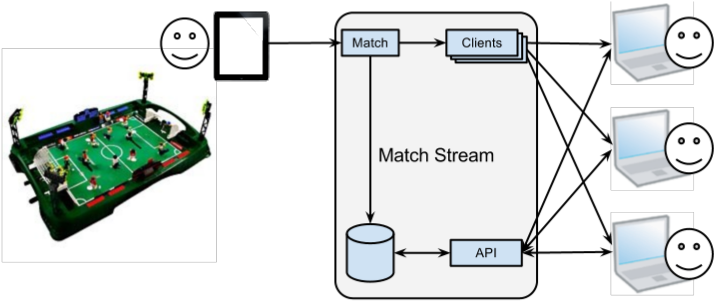
Match Stream
Requirements

- 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

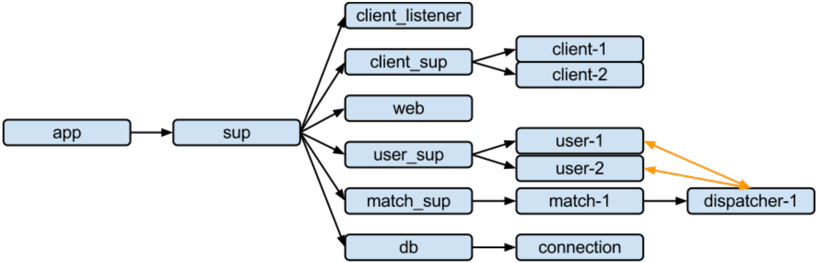
Match Stream
General Design



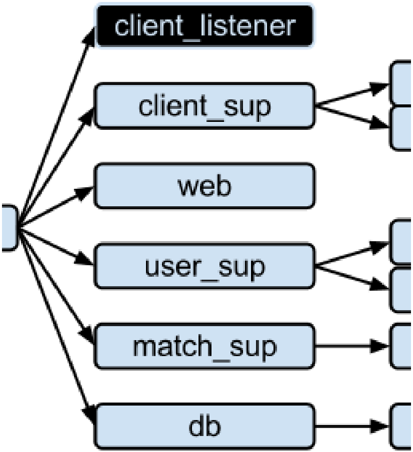
Match Stream
Sample

```
> telnet <server> <port>     |2011-09-13 13:48:51: goal:
...                          |  player: Luna (7)
Welcome to Match Stream.     |  team: <<"tig">>
...                          |
V:2:elbrujohalcon            |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

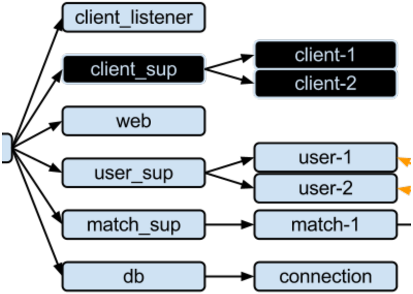


Components



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

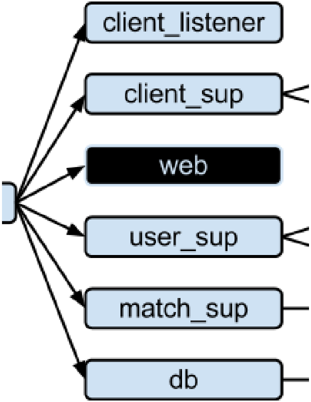
Components



CLIENT_SUP `supervisor`. Supervises connection processes

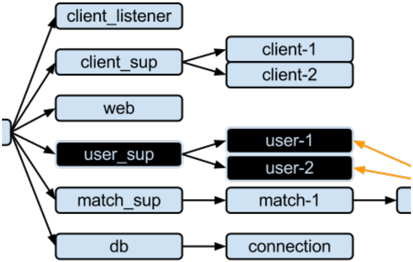
CLIENT `gen_fsm`. Handles a TCP connection

Components



WEB mochiweb server.
Listens for HTTP API calls

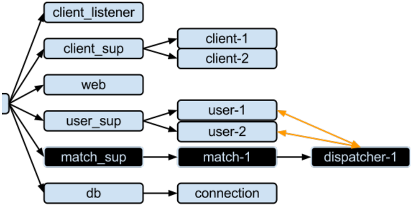
Components



USER_SUP `supervisor`. Supervises user processes

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

Components

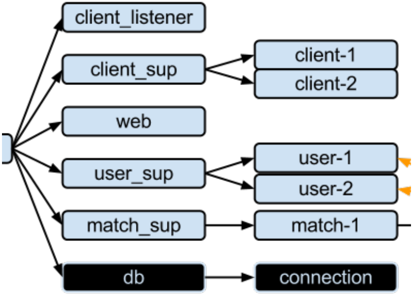


MATCH_SUP `supervisor`. Supervises match processes

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	**35s**

Stage 2.1
Connection Tweaks

Backlog

- Allow more concurrent connections
- Don't forget to TCP tune 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

- 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

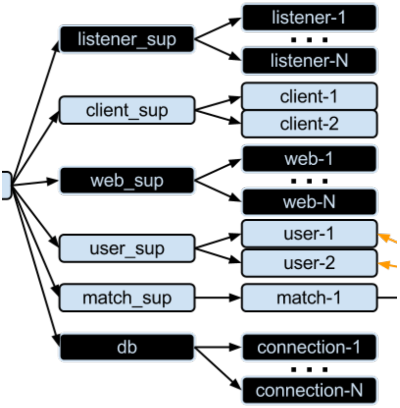
Listeners

- 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

- Distribute the work
- Use *repeaters*

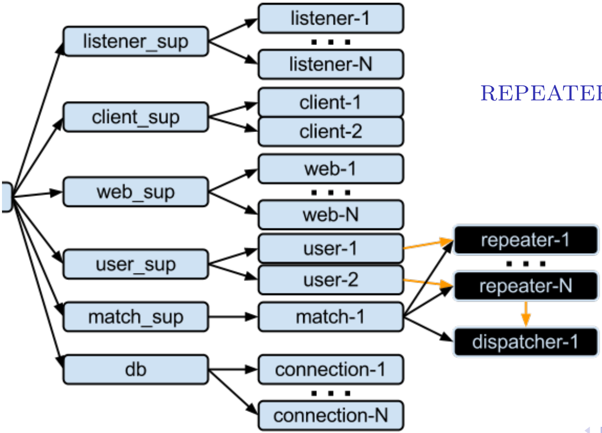
Stage 2.2

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	**1000**
ART	**2s**

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	**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}}.
```

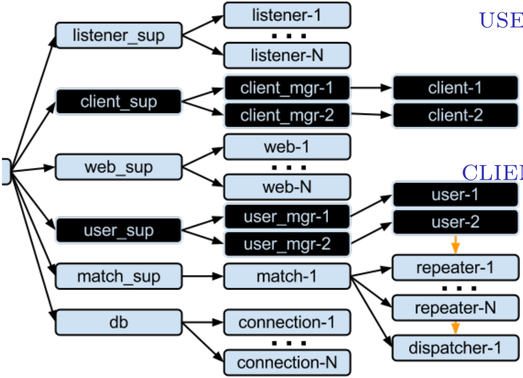
Stage 2.4

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

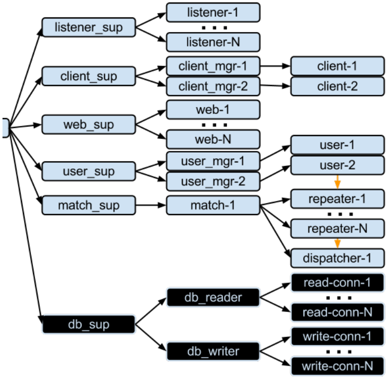
- Prepare the system to run in more than one node
- Decide if nodes should be connected or independent
- Decide if nodes should be on the same machine or not
- Determine which processes should be registered globally

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:		We scale up to:		Our improvements:	
N	**1K**	N	**100K**	N	**100x**
C	**4**	C	**32K**	C	**8000x**
ART	**26s**	ART	**10ms**	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:		We scale up to:		Our improvements:	
N	**1K**	N	**100K**	N	**100x**
C	**4**	C	**32K**	C	**8000x**
ART	**26s**	ART	**10ms**	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

Thanks!