# Scaling Erlang Web Applications

## 100 to 100K users at one web server

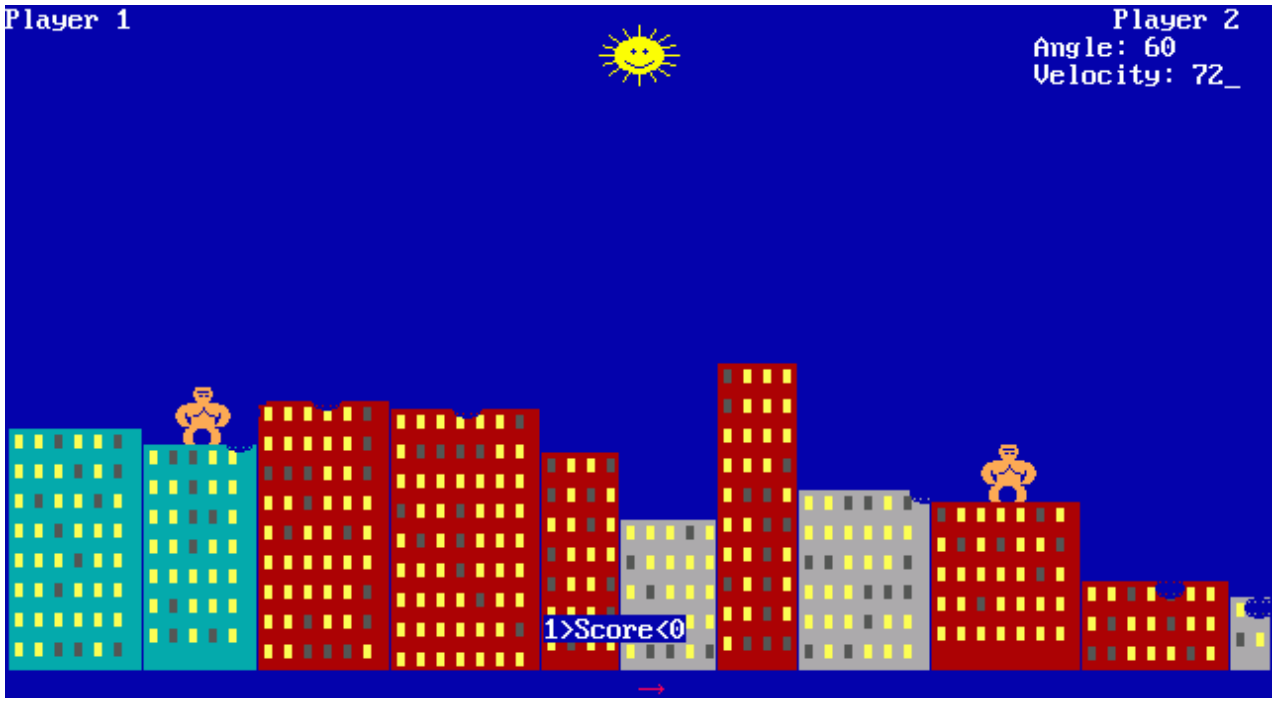
Fernando Benavides (*@elbrujohalcon*)

Inaka Labs

March 22, 2012

# Hello World!

- I'm a developer since I was 10
- I'm an Erlang developer since 2008




# Hello World!

- I'm a developer since I was 10
- I'm an Erlang developer since 2008

# Hello World!

- I've built several dynamic web servers
  - Many of them with real-time updates
  - Most of them with high scale requirements
- I'll show you how I make them scale

# Hello World!

- I've built several dynamic web servers
  - Many of them with real-time updates
  - Most of them with high scale requirements
- I'll show you how I make them scale

# Hello World!

- I've built several dynamic web servers
  - Many of them with real-time updates
  - Most of them with high scale requirements
- I'll show you how I make them scale

# Hello World!

- I've built several dynamic web servers
  - Many of them with real-time updates
  - Most of them with high scale requirements
- I'll show you how I make them scale

# Introduction

We will work 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.

Examples:

1. Social sites
2. Chat sites
3. Sports sites

# Introduction

We will work 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.

Examples:

1. Social sites
2. Chat sites
3. Sports sites

# Introduction

We will work 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.

Examples:

- Social sites
- Chat sites
- Sports sites

# Introduction

We will work 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.

Examples:

- Social sites
- Chat sites
- Sports sites

# Introduction

We will work 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.
Examples:

- Social sites
- Chat sites
- Sports sites

# Introduction

We will work 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.
Examples:

- Social sites
- Chat sites
- Sports sites

# Introduction

We will work 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.
Examples:

- Social sites
- Chat sites
- Sports sites

# Scope

*We will try to 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 try to 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

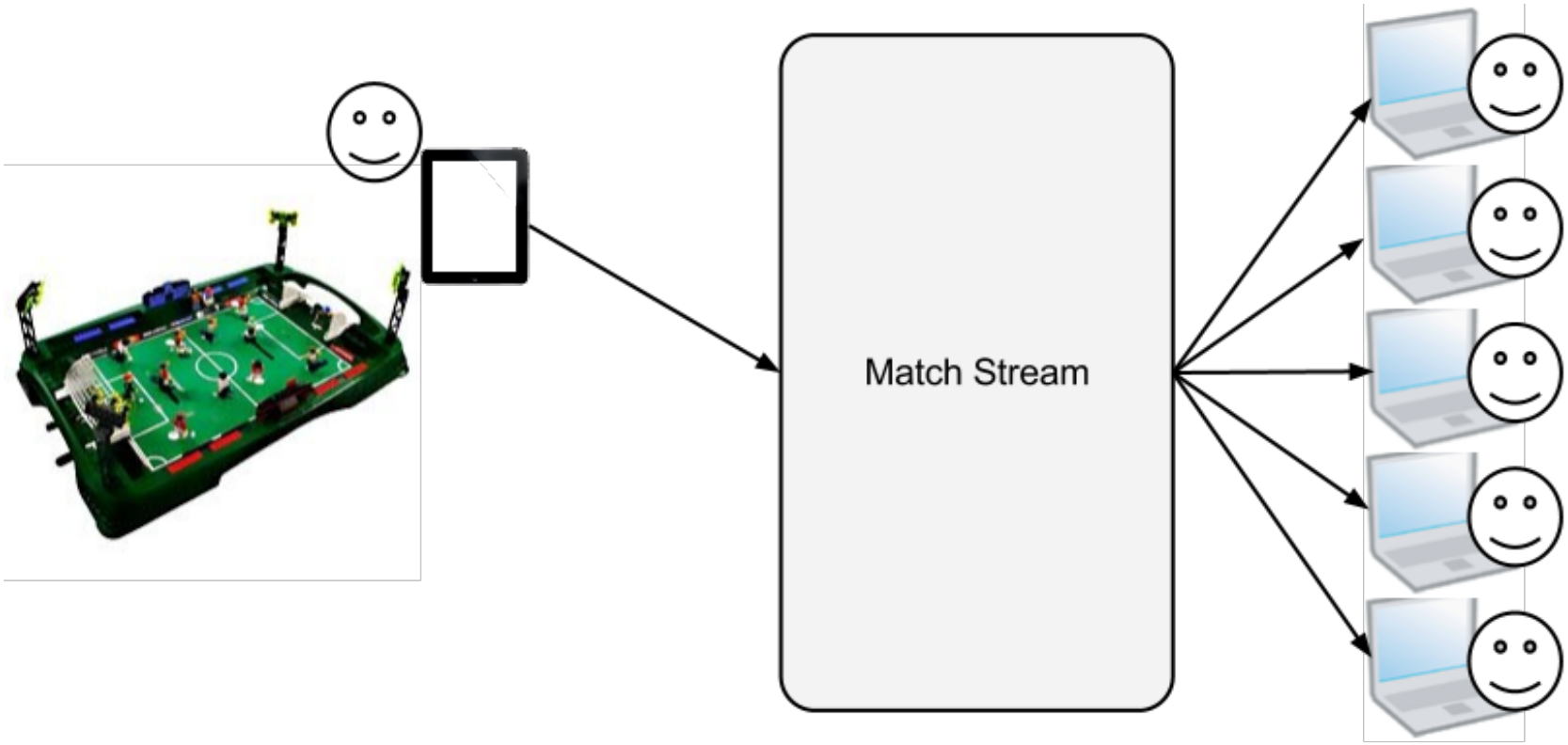
*Soccer fans are connected to the internet in their offices*

# 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

- Tons of concurrent users
- 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

- Tons of concurrent users
- 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

- Tons of concurrent users
- Two-hour-long bursts of connections followed by long periods of inactivity
- Real-time updates

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

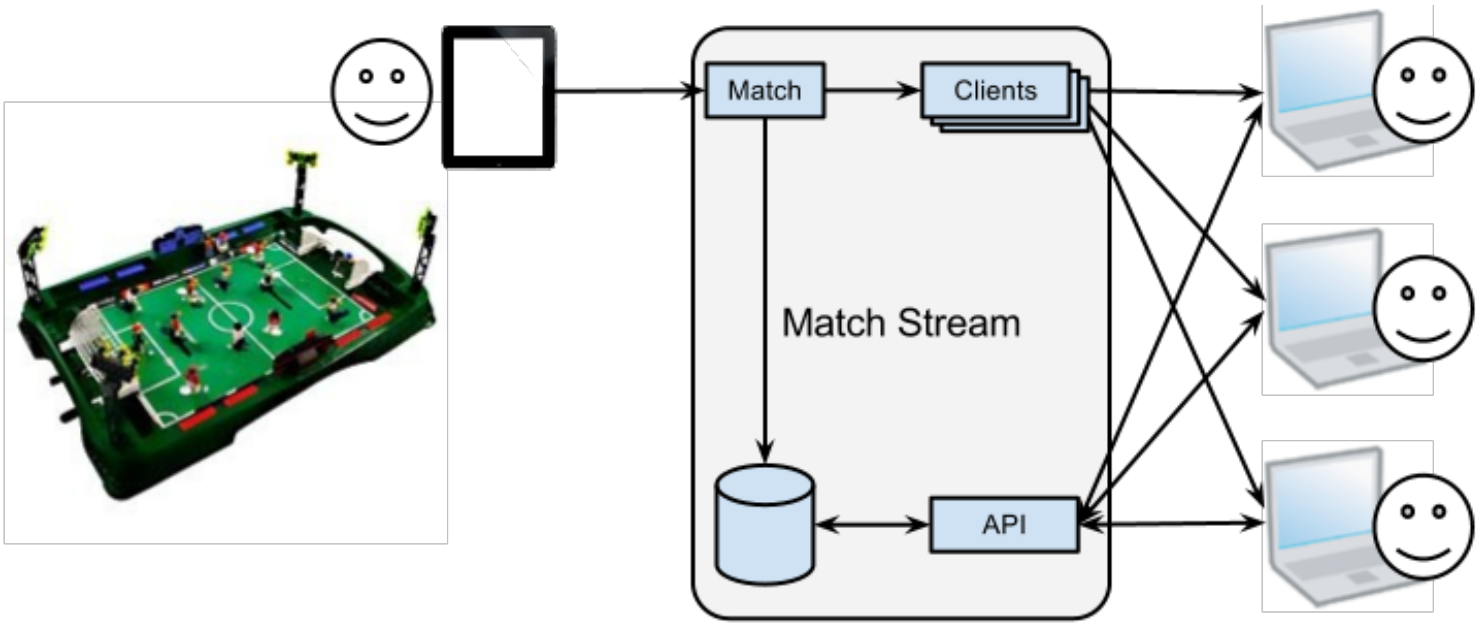
# Match Stream
## Requirements

### System Challenges

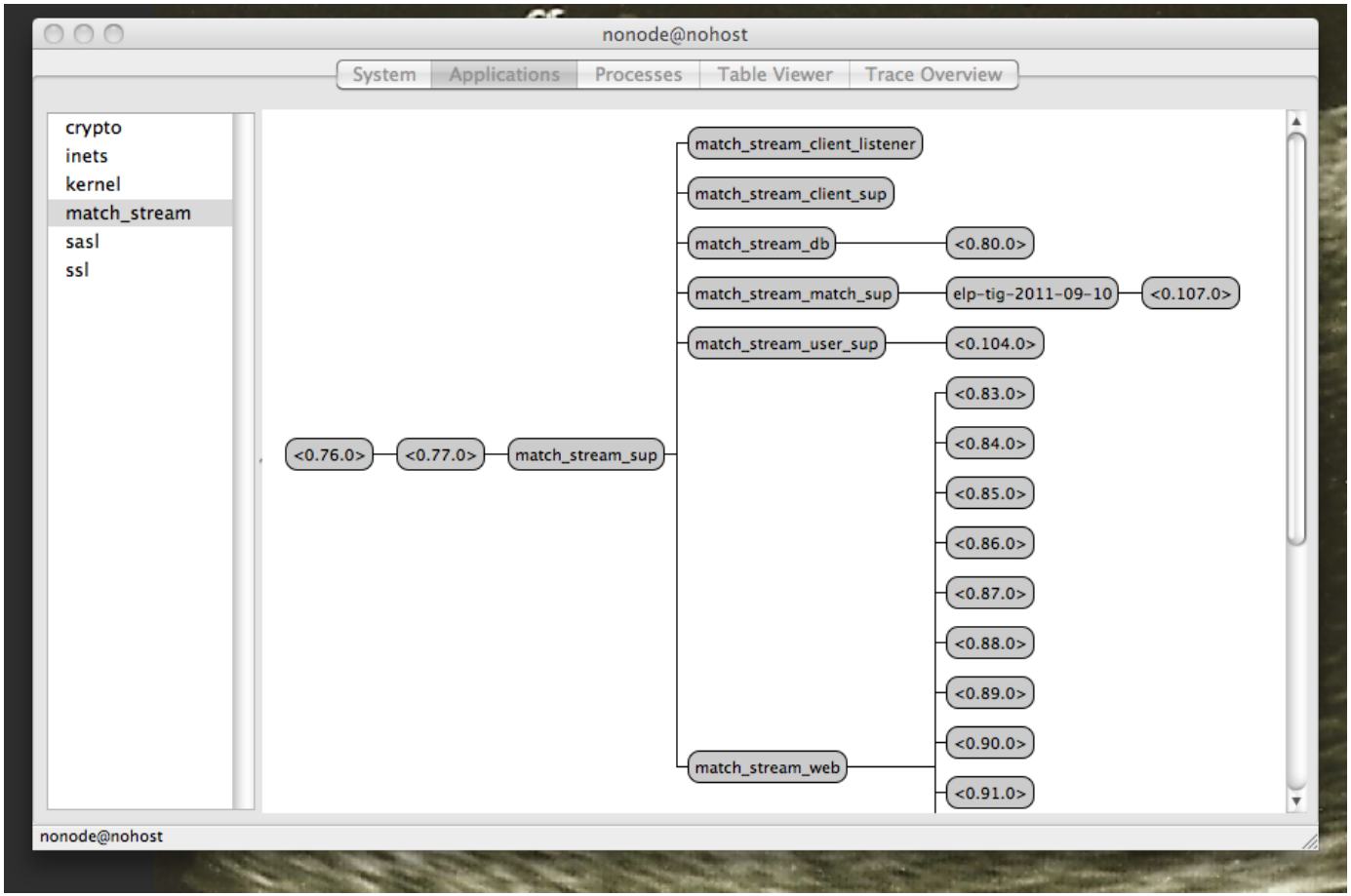
- Tons of concurrent users
- Two-hour-long bursts of connections followed by long periods of inactivity
- Real-time updates

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

# Match Stream
## General Design



# Match Stream
## Architecture



# Match Stream
## Client-handling Components

CLIENT_LISTENER

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

CLIENT_SUP

`supervisor`. Supervises connection processes

USER_SUP

`supervisor`. Supervises user processes

WEB

`mochiweb server`. Listens for HTTP API calls

# Match Stream
## Client-handling Components

CLIENT_LISTENER

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

CLIENT_SUP

`supervisor`. Supervises connection processes

USER_SUP

`supervisor`. Supervises user processes

WEB

`mochiweb server`. Listens for HTTP API calls

# Match Stream
## Client-handling Components

CLIENT_LISTENER

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

CLIENT_SUP

`supervisor`. Supervises connection processes

USER_SUP

`supervisor`. Supervises user processes

WEB

`mochiweb server`. Listens for HTTP API calls

# Match Stream
## Client-handling Components

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

CLIENT_SUP
: `supervisor`. Supervises connection processes

USER_SUP
: `supervisor`. Supervises user processes

WEB
: `mochiweb server`. Listens for HTTP API calls

# Match Stream
## DB and Watcher Components

DB

`gen_server`. Handles a connection to the DB

MATCH_SUP

`supervisor`. Supervises match processes

# Match Stream
## DB and Watcher Components

DB
: `gen_server`. Handles a connection to the DB

MATCH_SUP
: `supervisor`. Supervises match processes

# Lesson Learned

*Using Erlang to build your system is* ***not enough*** *to ensure* ***scalability***

# Stage 1
## Testing the system as it is

### 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 trying the system himself

# Stage 1
## Testing the system as it is

### Goals

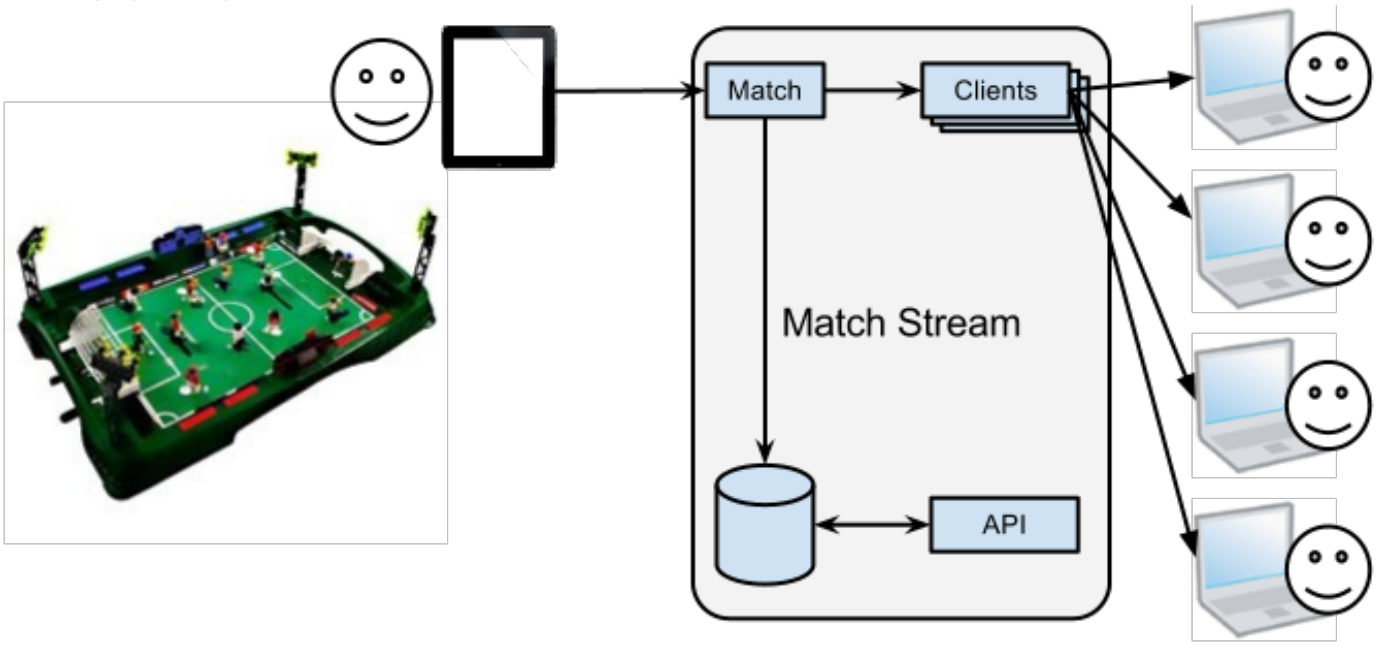
- Find how much the system can handle

### Steps

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

# Stage 1
## Testing the system as it is

### Results




**N = 1024 / C = 4**

# Stage 2
## Improving the Environment

### Goals

- Improve the system environment without altering the code

### Settings to Tune Up

- Concurrent TCP connections
- Open files limit
- TCP backlog size
- TCP memory allocation
- Erlang VM startup parameters

# Stage 2
## Improving the Environment

### Goals

- Improve the system environment without altering the code

### Settings to Tune Up

- Concurrent TCP connections
- Open files limit
- TCP backlog size
- TCP memory allocation
- Erlang VM startup parameters

# Stage 2
## Improving the Environment

### Goals

- Improve the system environment without altering the code

### Settings to Tune Up

- Concurrent TCP connections
- Open files limit
- TCP backlog size
- TCP memory allocation
- Erlang VM startup parameters

# Stage 2
## Improving the Environment

### Goals

- Improve the system environment without altering the code

### Settings to Tune Up

- Concurrent TCP connections
- Open files limit
- TCP backlog size
- TCP memory allocation
- Erlang VM startup parameters

# Stage 2
## Improving the Environment

### Goals

- Improve the system environment without altering the code

### Settings to Tune Up

- Concurrent TCP connections
- Open files limit
- TCP backlog size
- TCP memory allocation
- Erlang VM startup parameters

# Stage 2
## Improving the Environment

### Goals

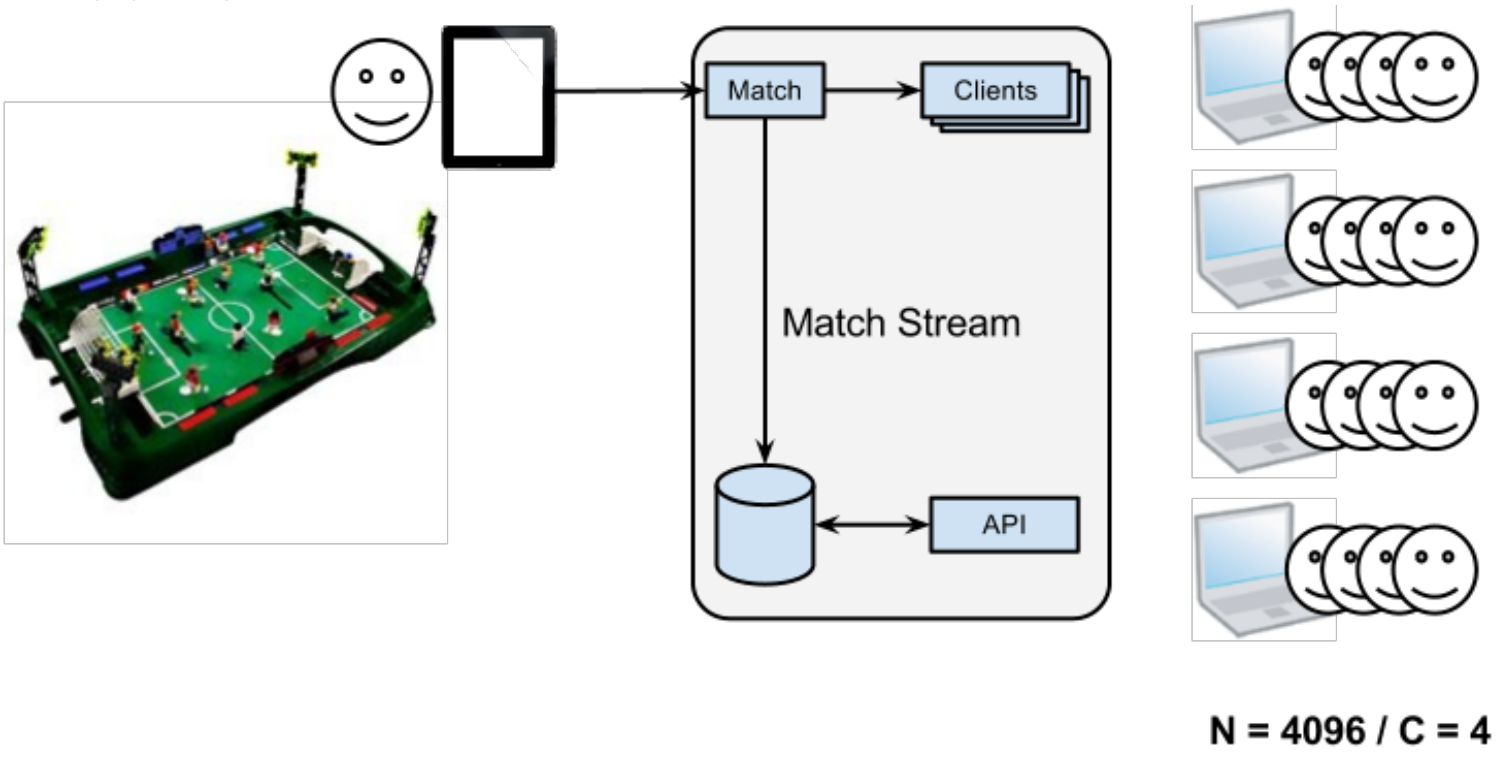
- Improve the system environment without altering the code

### Settings to Tune Up

- Concurrent TCP connections
- Open files limit
- TCP backlog size
- TCP memory allocation
- Erlang VM startup parameters

# Stage 2
## Improving the Environment

### Results



# Stage 3
## Improving Match Stream

### Goals

- Tune up the system for one node

### Steps

- Find a problem
- Fix it using the list of *Tips and Tricks*
- If not there, add it
- Repeat from Stage 1

# Stage 3
## Improving Match Stream

### Goals

- Tune up the system for one node

### Steps

- Find a problem
- Fix it using the list of *Tips and Tricks*
- If not there, add it
- Repeat from Stage 1

# Stage 3.1
## Connection Tweaks

### Backlog

- Allow more concurrent connections
- Remember HTTP *runs on* TCP

### Connections

- Don't use just one of them
- Check inbound and outbound connections

# Stage 3.1
## Connection Tweaks

### Backlog

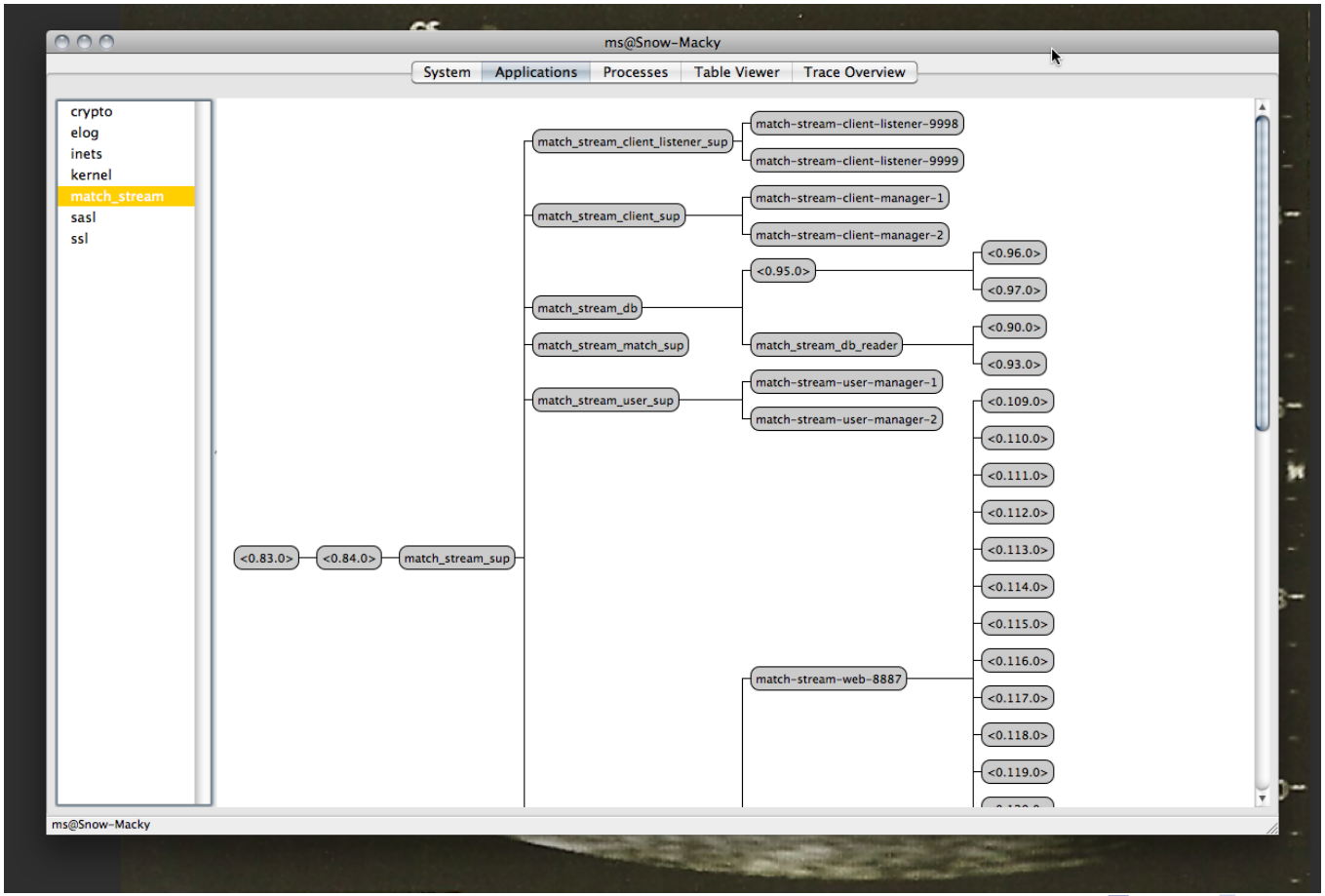
- Allow more concurrent connections
- Remember HTTP *runs on* TCP

### Connections

- Don't use just one of them
- Check inbound and outbound connections

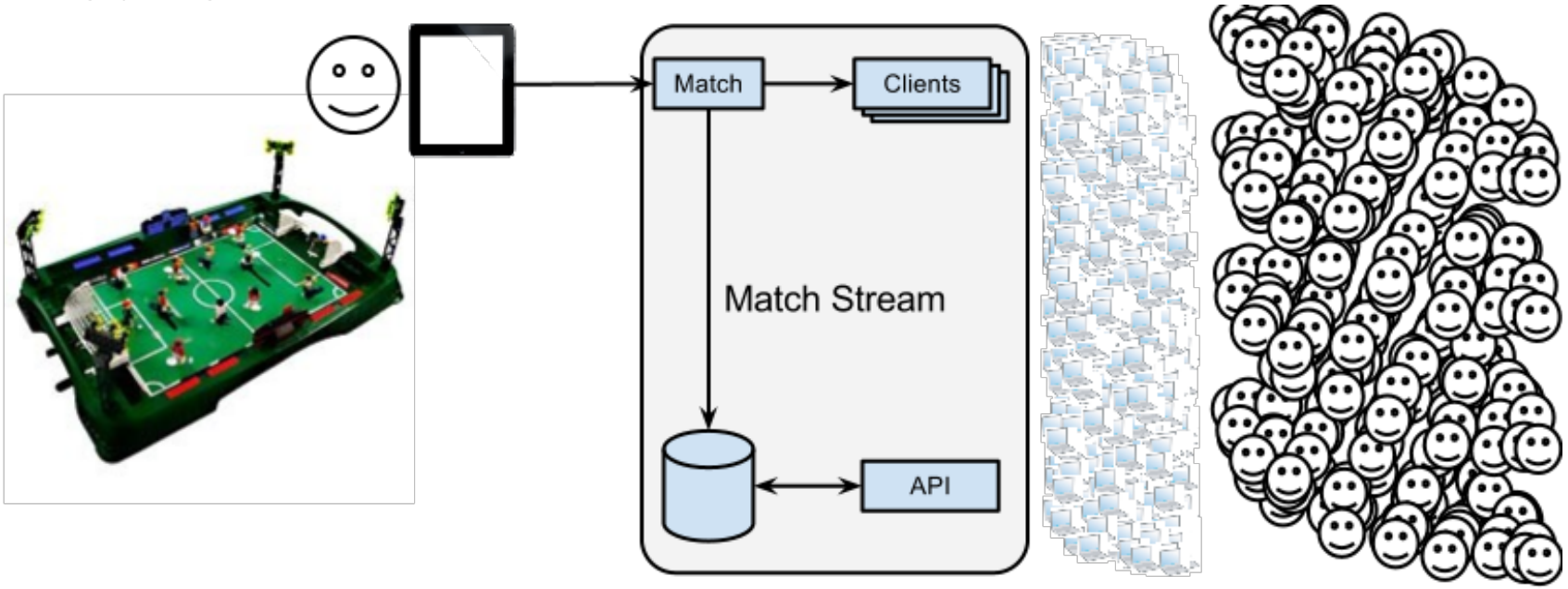
# Stage 3.1
## Connection Tweaks

## System Architecture



# Stage 3.1
## Connection Tweaks

### Results




N = 65536 / C = 8192

# Stage 3.2

## gen_event

### sup_handler

- Don't use it
- Monitor the processes instead

### Long Delivery Queues

- Use *repeaters*

# Stage 3.2
## gen_event

### sup_handler

- Don't use it
- Monitor the processes instead

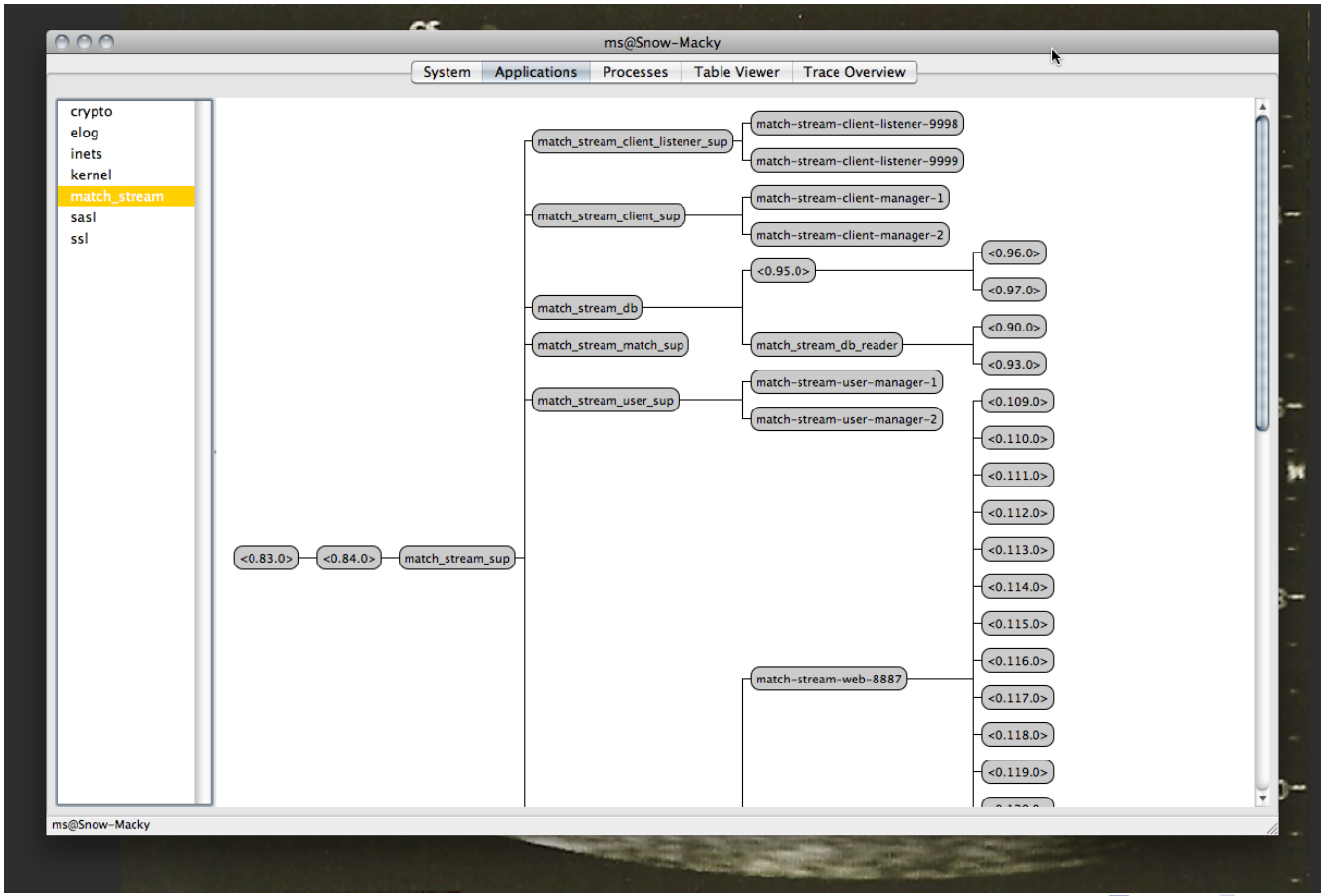
### Long Delivery Queues

- Use *repeaters*

# Stage 3.2

## gen_event

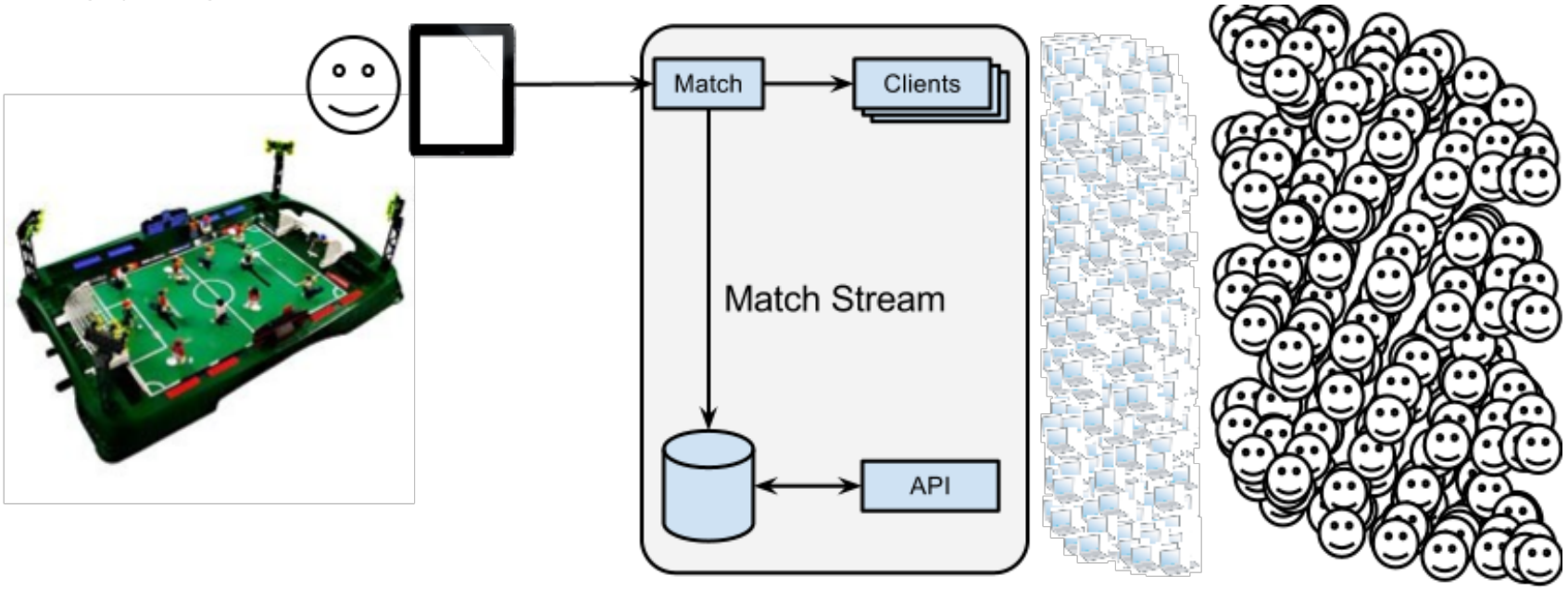
## System Architecture




# Stage 3.2

## gen_event

### Results




N = 65536 / C = 8192

# Stage 3.3
## gen_server

### Call Timeouts
Remember `gen_server:reply/2`

### Memory Footprint
Remember `hibernate`

### Long init/1
Use 0 timeout

# Stage 3.3

## gen_server

### Call Timeouts

Remember `gen_server:reply/2`

### Memory Footprint

Remember `hibernate`

### Long init/1

Use 0 timeout

# Stage 3.3

## gen_server

### Call Timeouts

Remember `gen_server:reply/2`

### Memory Footprint

Remember `hibernate`

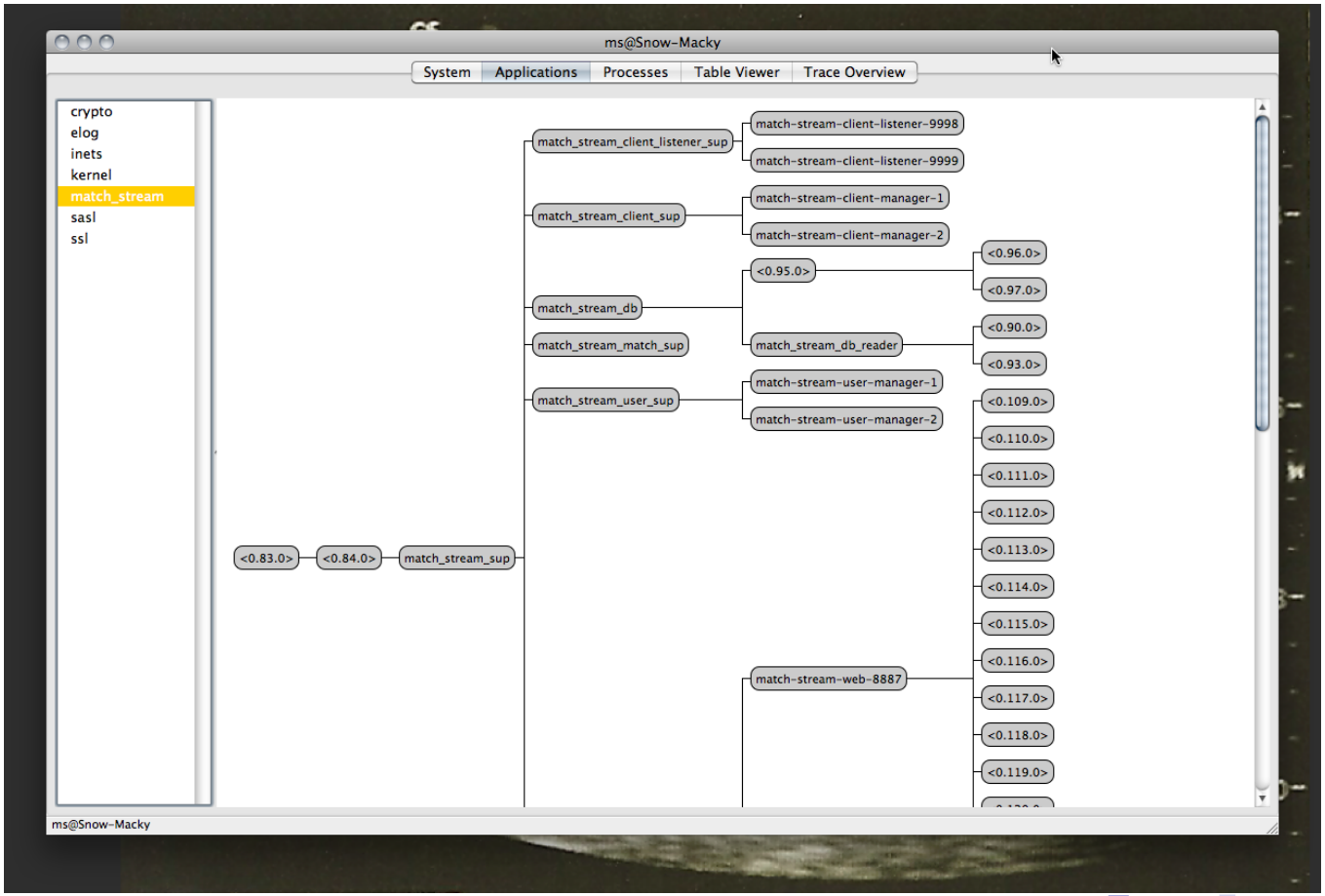
### Long init/1

Use 0 timeout

# Stage 3.3

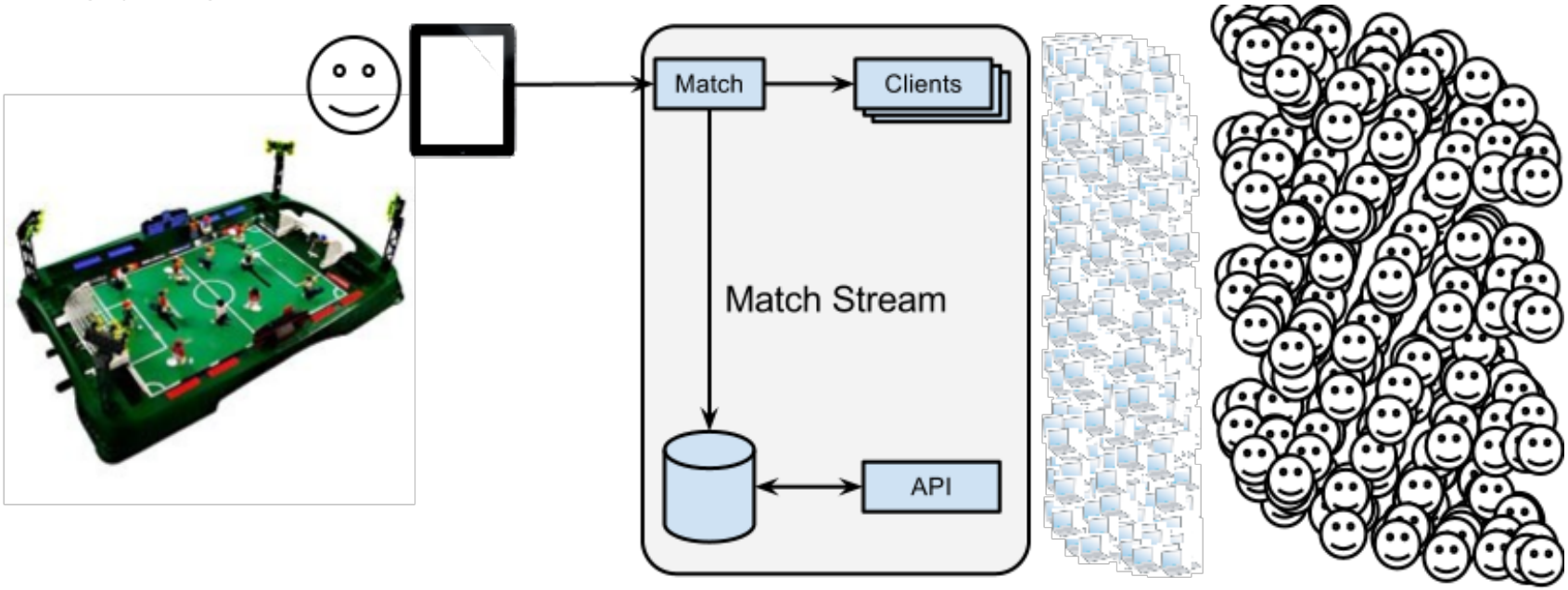
## gen_server

## System Architecture




# Stage 3.3
## gen_server

### Results




N = 65536 / C = 8192

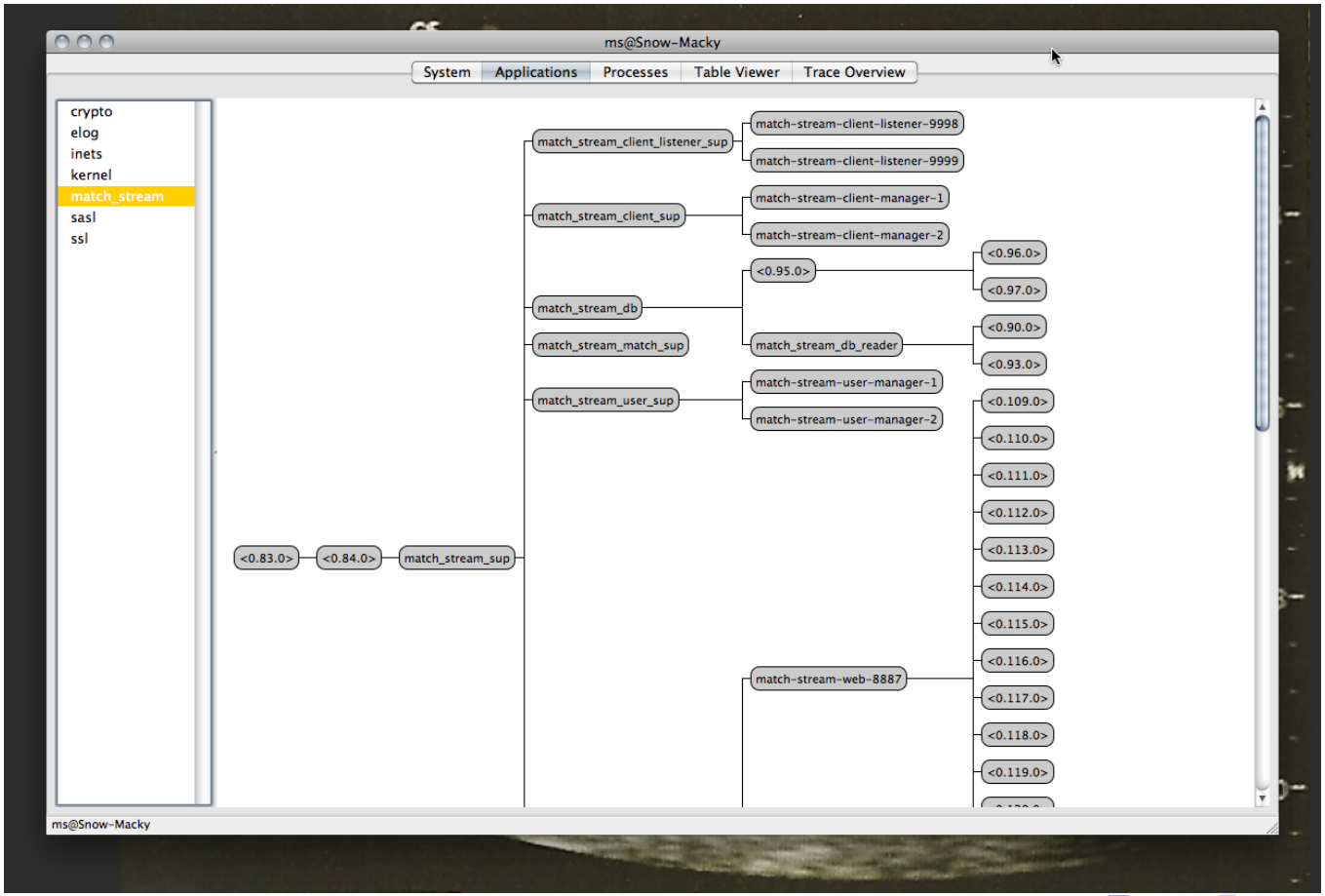
# Stage 3.4
## supervisors

- 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(random:uniform(#ofSupervisors))).
```

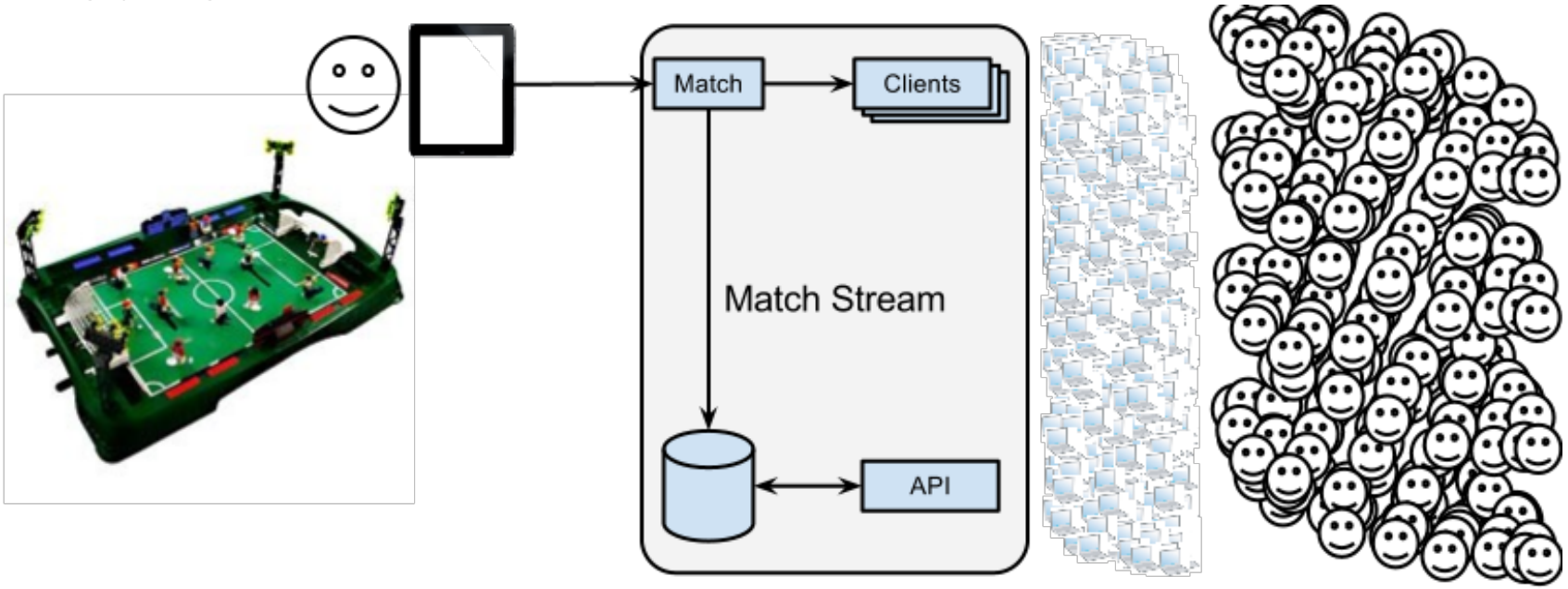
# Stage 3.4
## supervisors

## System Architecture




# Stage 3.4
## supervisors

### Results




N = 65536 / C = 8192

# Stage 3.5
## 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 3.5
## 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 3.5
## Other Processes

### Timers

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

### Logging

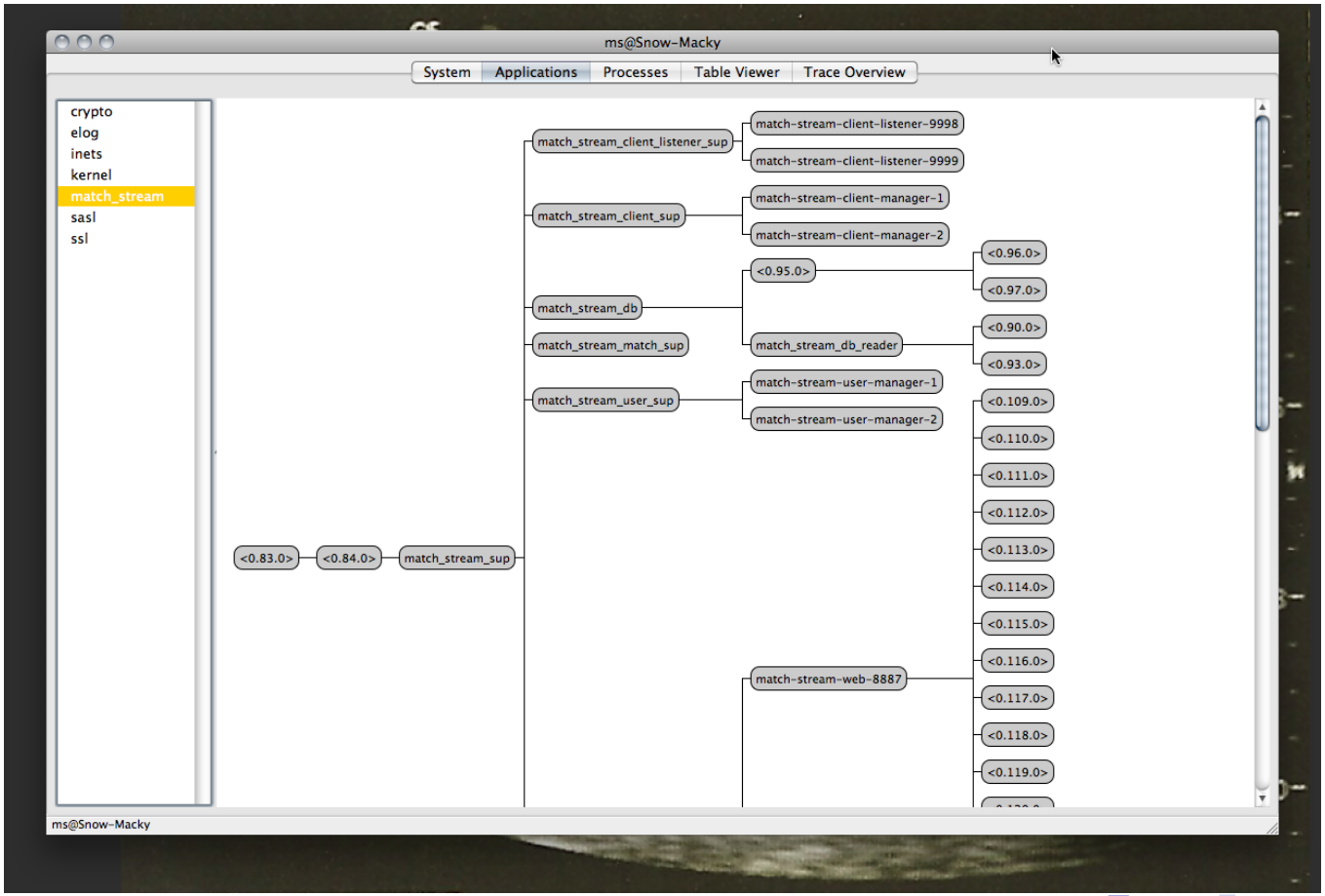
- Don't log too much
- Use a good logging system

### 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

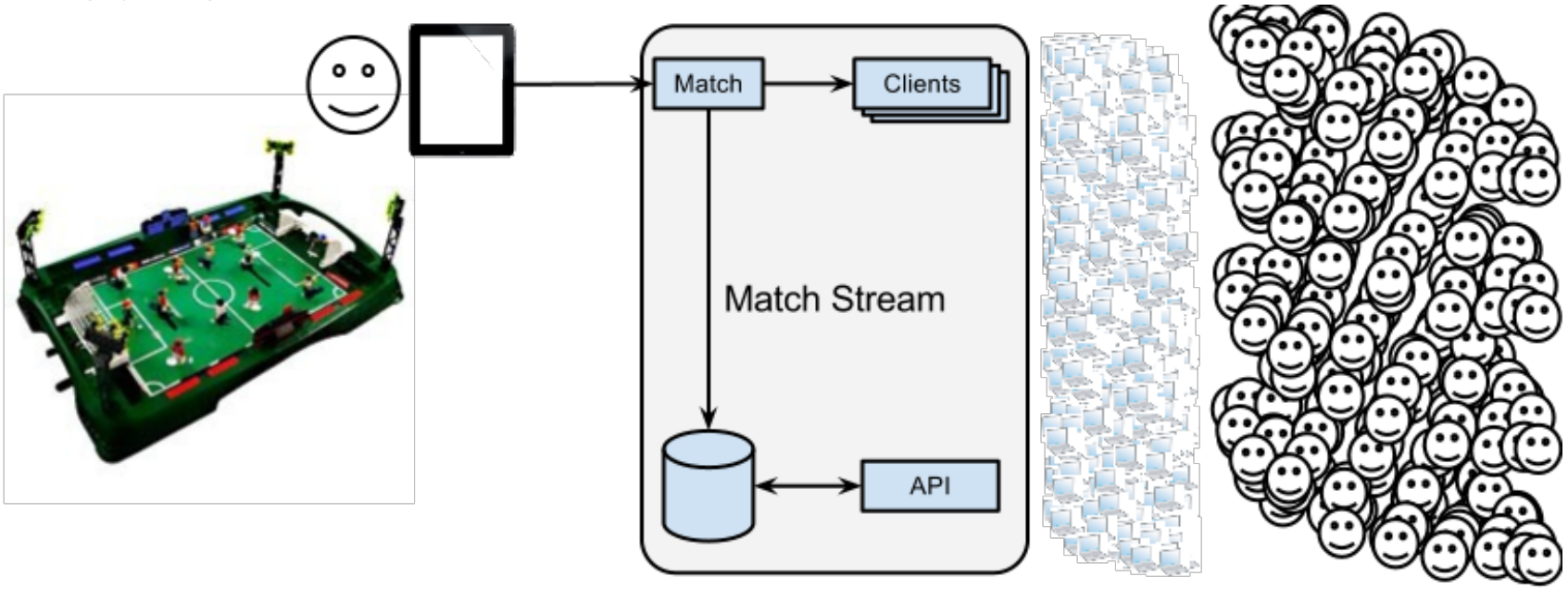
# Stage 3.5
## Other Processes

## System Architecture




# Stage 3.5
## Other Processes

### Results




N = 65536 / C = 8192

# Stage 4
## 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 4
## Adding Nodes

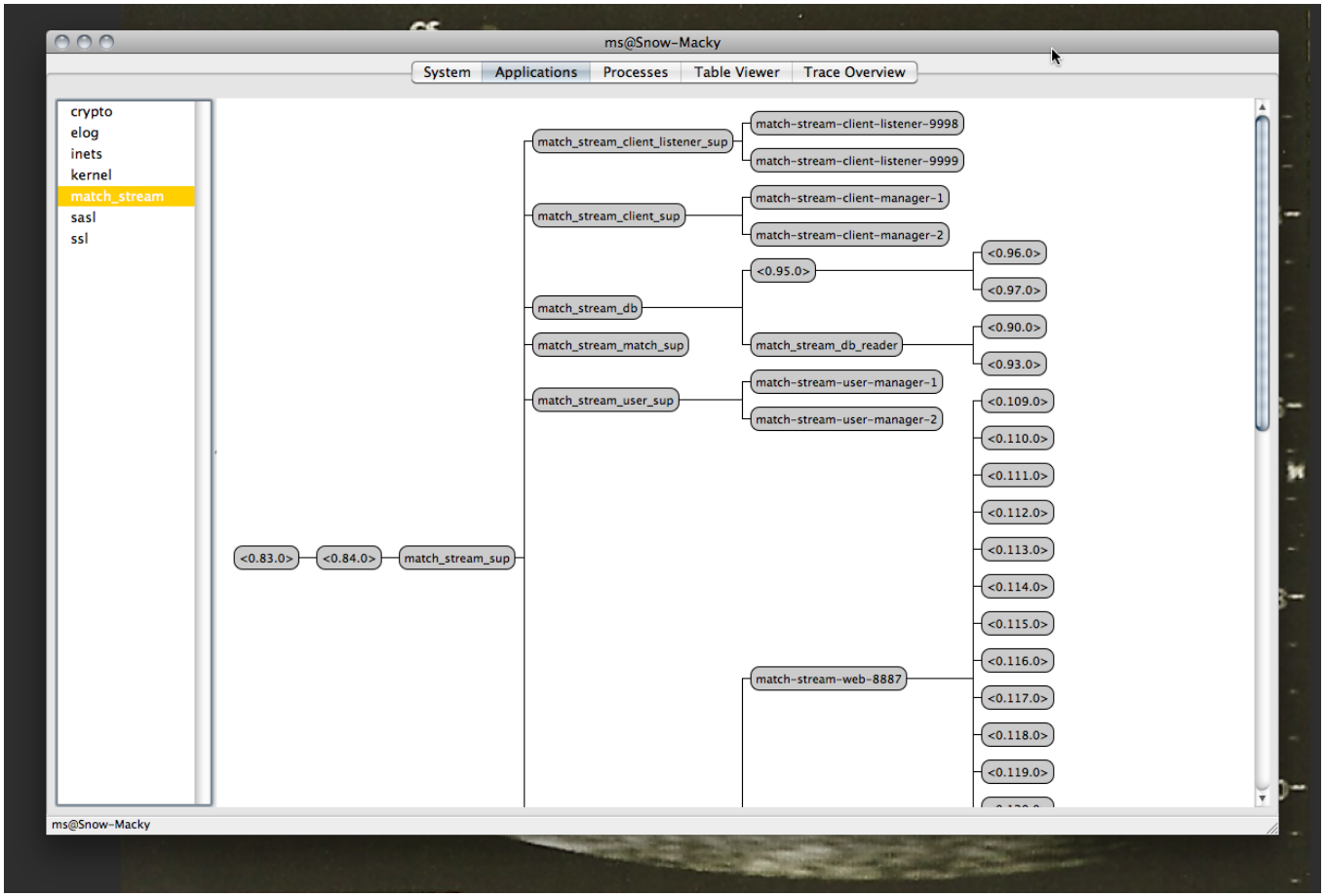
### Goals

- 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

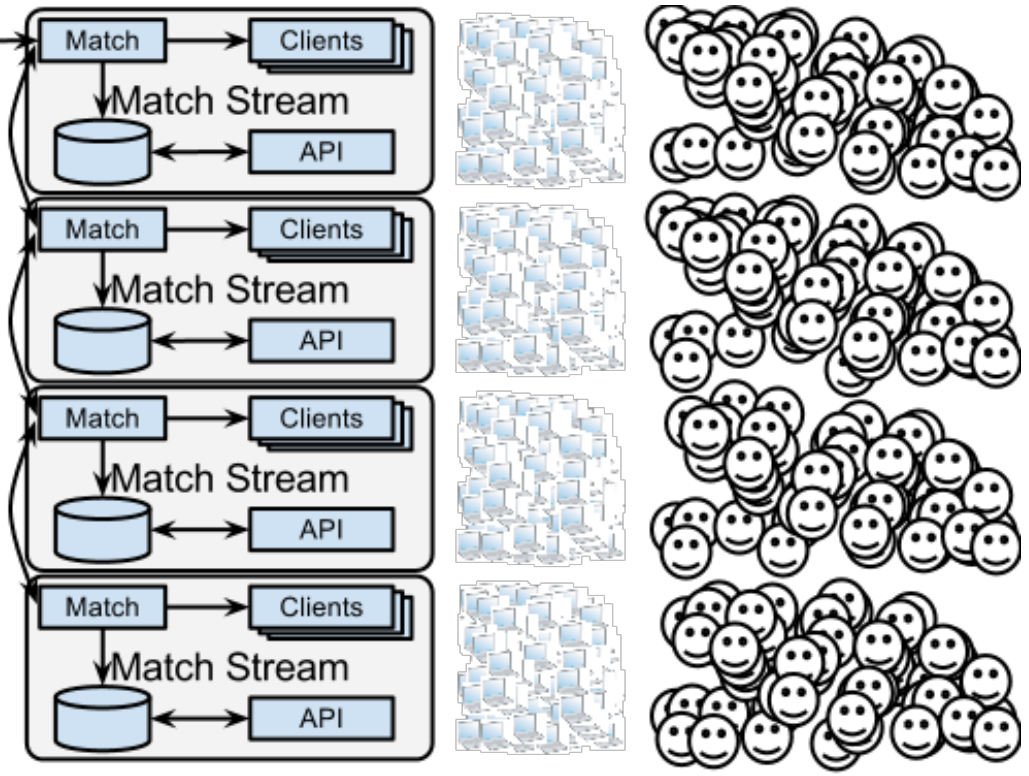
# Stage 4
## Adding Nodes




# Stage 4
## Adding Nodes

### Results

N ≈ 25000 * 4 ≈ 100000
C ≈ 8192 * 4 ≈ 32768



# 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!