# Controlling Datacenter Cooling Infrastructure

Erlang Factory SF 2014
Garret Smith
garret.smith@gmail.com

SYNAPSENSE

Discriminating Erlang hacker
Did not make video
Garret Smith

Discriminating Erlang hacker
Made “OTP 2” video
The “other” Garret Smith

SYNAPSENSE

# Outline

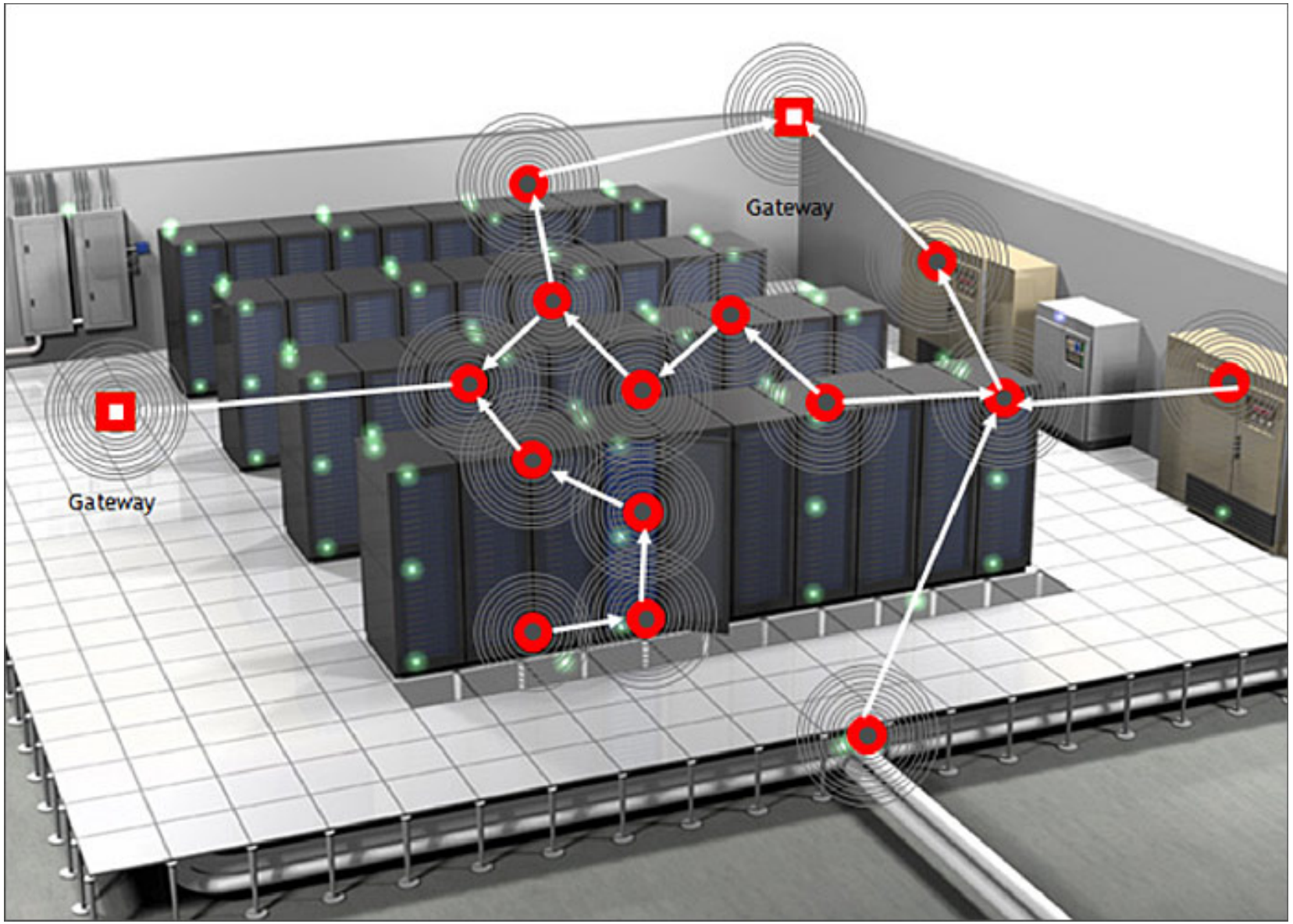
- Unique Erlang application
- Java / Erlang integration with jinterface
- Failover design
  - Active/passive to horizontally scalable
- Designing for reliability
  - Parallels in mechanical infrastructure

SYNAPSENSE

# Company history

- SynapSense formed to bring wireless mesh networking to market
- Application: environmental monitoring in datacenters
  - Huge energy budget cooling expensive assets (your cloud servers)
  - Increasing energy (and carbon footprint) awareness
  - Energy savings & reliability with real-time environmental monitoring

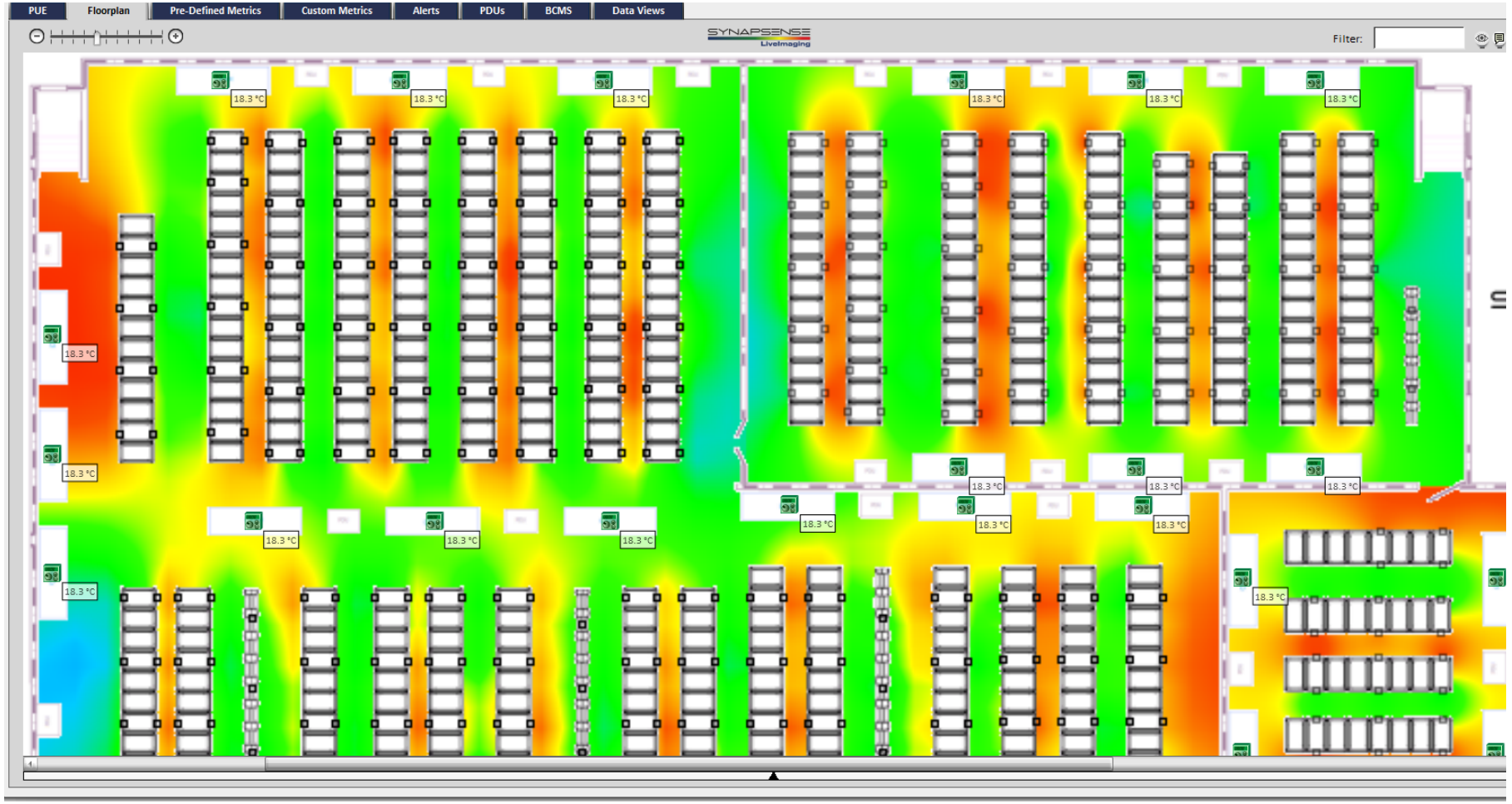
SYNAPSENSE®




# What we do

SYNAPSENSE

# Lots of analytics & visualization




SYNAPSENSE

- Add intelligence to cooling infrastructure
  - Think motion-activated lights on a much larger scale
- Better efficiency – energy savings
- Better reliability – fewer server failures

**Analytics good, $$$ better**

SYNAPSENSE

Room geometry
Comm details

Configuration data

Sensor data stream

Effects of adjustment propagate through environment

Control system

Device I/O

SYNAPSENSE

- Isolate control function
- Embedded vs server-class
- Redundancy

## How to build a reliable controller?

SYNAPSENSE

| Language | GC | Interactive Shell | Cross Platform | Concurrency | HA | Distributed DB | Linux/ARM | Hot reload | Realtime |
|---|---|---|---|---|---|---|---|---|---|
| C | no | no | partial | library support | N/A | 3rd party | yes | no | hard |
| C++ | no | no | partial | library support | N/A | 3rd party | yes | no | hard |
| Java | yes | no | yes | library support | app server | 3rd party | yes | no | no |
| Python | yes | Ipython | yes | Ipython | N/A | 3rd party | yes | yes | no |
| Erlang | yes | yes | yes | language level | Standard | Standard | yes | yes | soft |
| C# | yes | no | no | library support | app server | 3rd party | no | no | no |

Circa 2009

# Why Erlang?

SYNAPSENSE

# Introducing Erlang to a Java system

SYNAPSENSE®

- Start simple: zero impact to Java services
- Refine Java interface, managed by Erlang
- Merge code into Java service for efficiency

# Easing into Erlang

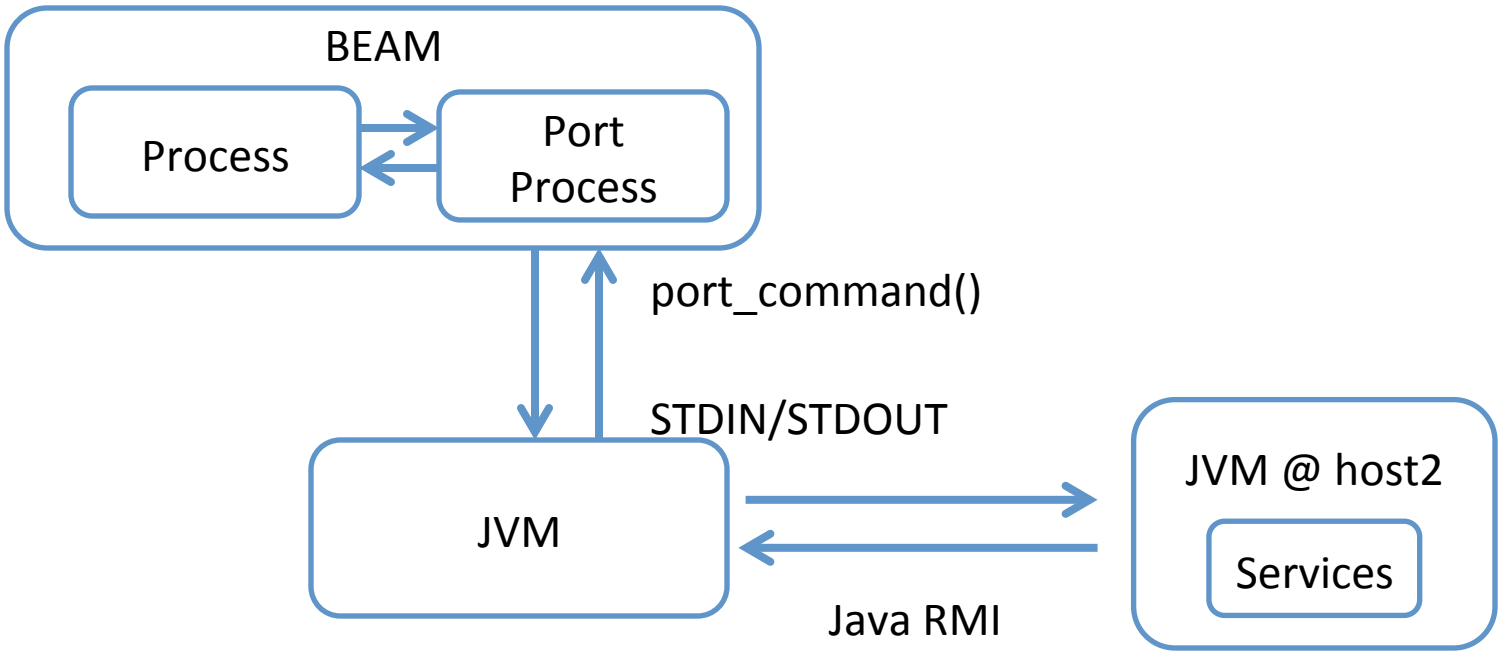
SYNAPSENSE

# Port driver

- Launch Java program for RMI

```
open_port(
        {spawn, “java –jar rmi.jar”},
        [{packet, 4}, hide])
```

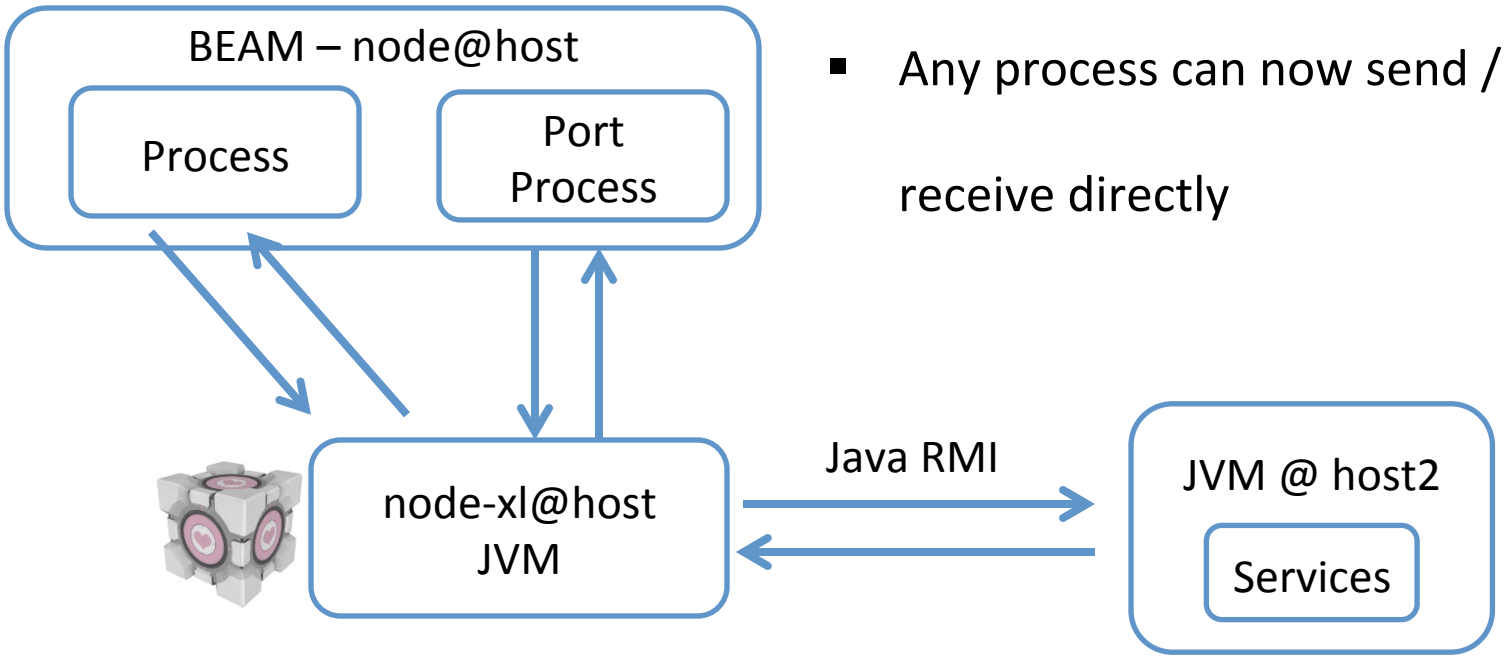
- Jinterface for marshaling Erlang terms




SYNAPSENSE

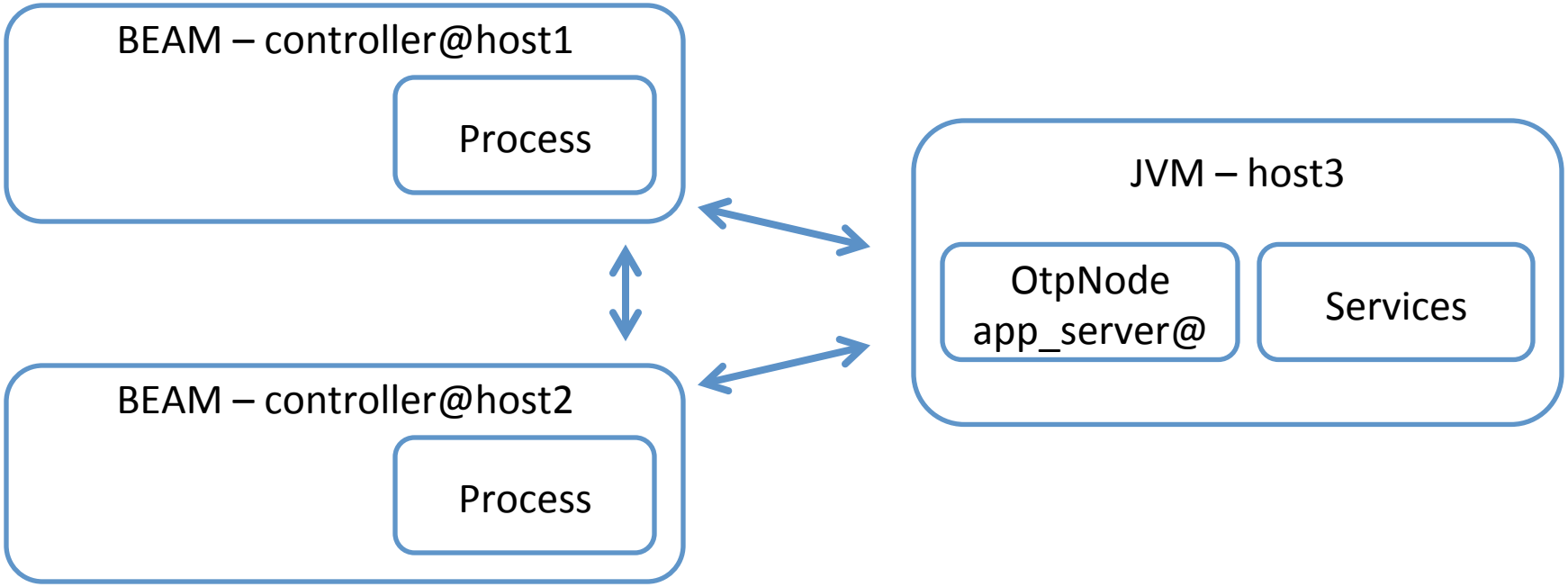
# Companion node

- TCP sockets instead of STDIN/ STDOUT
- Remove single-process bottleneck
  - since remedied in R16
- Any process can now send / receive directly




SYNAPSENSE

# Integrate with Java services



# Now for some code!

(snippets)

# Start the Java node

```
Command = “java –jar app.jar ” ++ NodeName,
PortPid = open_port(
            {spawn, CommandString},
            [{cd, PrivDirectory}, stream, exit_status, hide]),
% net_adm:ping(NodeName) until ‘pong’
erlang:monitor_node(NodeName, true),
erlang:send({javaproc, NodeName}, link)
```

```
public static void main(String[] args) {
    OtpNode node = new OtpNode(args[0]);
    node.setCookie("test");
    OtpMbox mbox = node.createMbox("javaproc");
    OtpMsg link_message = mbox.receiveMsg(60 * 1000);
    OtpErlangPid fromPid = link_message.getSenderPid();
    OtpErlangObject link = link_message.getMsg();
    if("link".equals(link.toString()) {
        mbox.link(fromPid);
    } else {
        System.exit(-1);
    }
    // . . .
}
```

SYNAPSENSE

# Flexible dispatching in Java

```
call(Function, Args) ->
    Ref = make_ref(),
    {javaproc, NodeName} ! {Ref, Function, Args},
    receive
        {Ref, Result} -> Result
    end.
```

```
OtpMsg m = mbox.receiveMsg();
OtpErlangPid fromPid = m.getSenderPid();
OtpErlangTuple callWrapper = (OtpErlangTuple) m.getMsg();
OtpErlangRef replyRef = (OtpErlangRef) callWrapper.elementAt(0);
OtpErlangAtom function = (OtpErlangAtom) callWrapper.elementAt(1);
OtpErlangList args = (OtpErlangList) callWrapper.elementAt(2);
```

```
executorService.execute(new ApiFunction(mbox, fromPid, replyRef, function, args));
```

SYNAPSENSE

# Class per API method

```
class ApiMethod implements Runnable {
    private OtpMbox mbox;
    private OtpErlangPid fromPid;
    private OtpErlangRef replyRef;
    private OtpErlangAtom function;
    private OtpErlangList args;

    public ApiMethod(
                            final OtpMbox mbox,
                            final OtpErlangPid fromPid,
                            final OtpErlangRef replyRef,
                            final OtpErlangAtom function,
                            final OtpErlangList args) {
        this.mbox = mbox;
        this.fromPid = fromPid;
        this.replyRef = replyRef;
        this.function = function;
        this.args = args;
    }
    public void run() {
        OtpErlangObject resp = args;  // Actual logic here, build response
        OtpErlangTuple taggedReply = new OtpErlangTuple(
                            new OtpErlangObject[] {replyRef, resp});
        mbox.send(fromPid, taggedReply);
    }
}
```

SYNAPSENSE

# Failover evolution

From active/standby to horizontally scalable

SYNAPSENSE

- Active/standby design
- Run mnesia on both servers
  - Cache config & process state
  - Active node pushes to standby
- Configure app with dist_ac
  - ‘dist_ac’ part of Erlang networking kernel
  - Keeps 1 instance running across a cluster
- Reload from cache for a warm start-up

**Start simple**

SYNAPSENSE

# Then we had our first netsplit

- Same behavior as dist_ac, except:
  - Recognize 2 running copies and stop 1
  - Handle abnormal application exit differently
- 1-process application (gen_leader)
- Starts/stops other application (ours)
- Application follows “leader”

**Replace dist_ac**

SYNAPSENSE

- Processes are the authoritative source
  - Data just to resurrect a process (tree)
- Only using memory tables
- 1 process – 1 row in table

# What to do about mnesia?

- Join the group
- Broadcast writes
- Cache received tuples

## pg2 - distributed process groups

```
write(Key, Data) ->
[Pid ! {Key, Data} || Pid <- pg2:get_members(proc_cache)].
```

SYNAPSENSE

- Use gen_leader processes internally
- Start our app on all nodes
- Distribute processes across nodes
- Less to migrate when something fails
- Faster & less disruptive failover

# Scaling out

# Designing for reliability

Parallels in mechanical infrastructure

SYNAPSENSE®

- What could fail?
- What is *likely* to fail?
- What can you do about it?

**Design for failure**

SYNAPSENSE

# N+1

SYNAPSENSE


# 2N

SYNAPSENSE

- Peers monitoring each other
  - Servers in active/passive configuration
  - Linked processes
- High-level / low-level
  - Our system monitoring air handlers
  - Supervisors monitoring processes
- Autonomy
  - Air handlers continue to operate if we fail
  - Process failures localized

# Systems monitoring systems

SYNAPSENSE

## Recap

- dist_ac & mnesia don’t like netsplits
  - Path not taken: github.com/uwiger/unsplit
- gen_leader can help
  - github.com/garret-smith/gen_leader
- So can pg2, but for different scenarios
- Reliability: design for failure

SYNAPSENSE

garret.smith@gmail.com

@GarretESmith

erlang-questions mailing list

**Thank you for coming!**

SYNAPSENSE