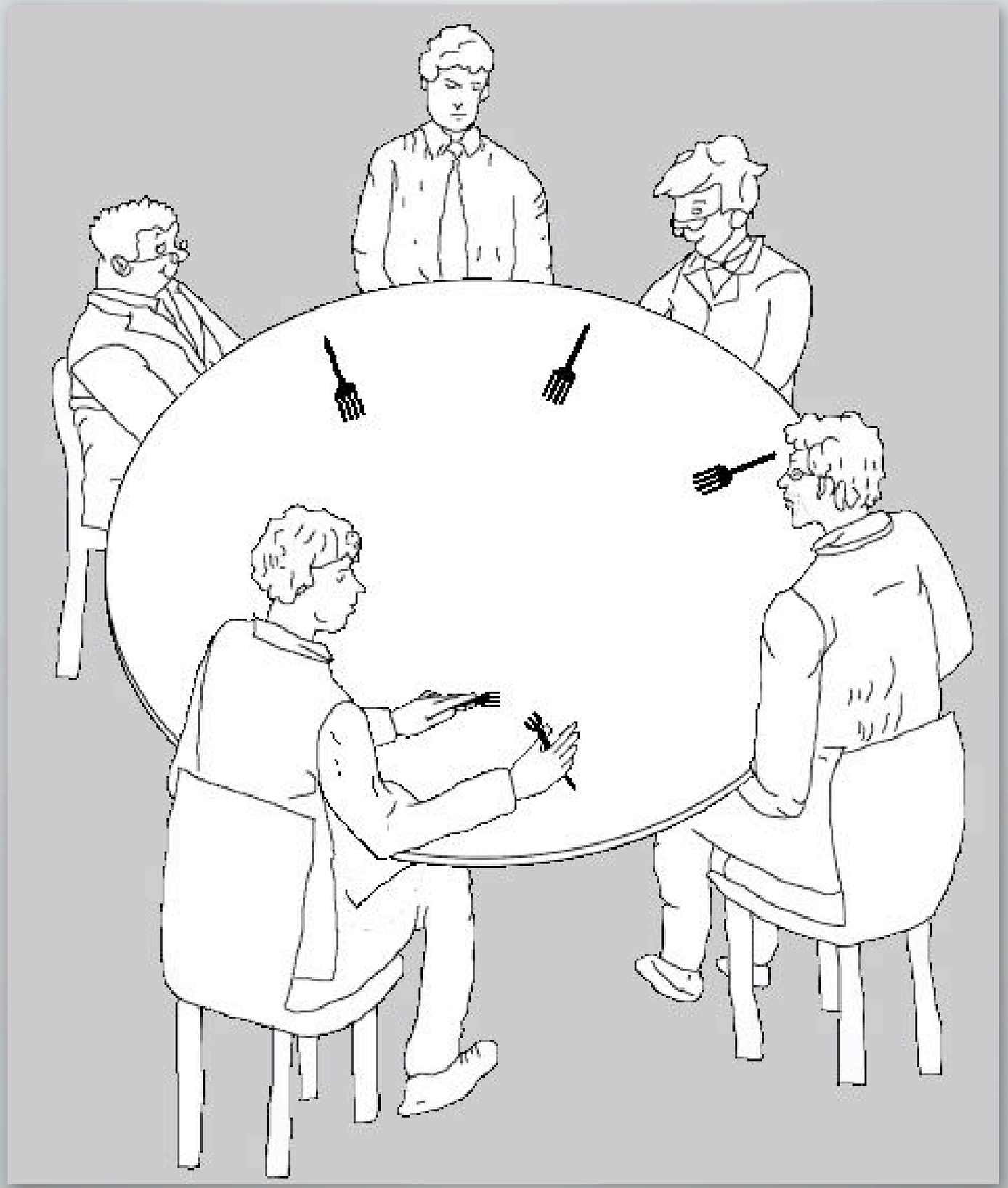
Erlang and Concurrency

André Pang
Rising Sun Research

rsr risingsun [RESEARCH]

Games. Unreal 3 engine. Amazing graphics.

(Unreal 3 video demo.) 2004/2005? Still state-of-the-art.



What does that game have to do with this talk? Graphics is an ‘embarrassing parallel’ problem. Speed increases due to sheer parallelism (pipelines on modern graphics cards). Who’s programmed with concurrency before? Hands up…

Though an Unreal Engine 3 game can easily consume 512MB memory (on console) or up to 2GB (on PC, in the editing tools), less than 10% of that memory consists of data that contains pointers. The rest is bulk binary resources - like texture maps, sounds, animations, which don't require scanning for references.

50MB of data is well within the range of a mark-and-sweep garbage collector taking a few milliseconds within a game running at 30 frames per second. This is what we do. Realtime garbage collection is another possibility, but we didn't want to try to implement that within a primarily C++ engine.

Garbage collecting memory:

I do believe this is completely practical as the sole memory management solution, even in a realtime application like a game.

Tim Sweeney

Sweeney is brilliant. John Carmack is no real comparison to Sweeney's design and architecture skills (though he probably beats him in sheer graphics wizardry.) Note: Erlang has a real-time garbage collector.

Second, there are the equivalance properties of values involving name binders. There are a number of possibilities for implementing this in a more general programming language. You could write a "mod-alpha-conversion" equivalance test that operates on such terms, treating this as a wider relation than equality (where some alpha-equivalant terms would be considered unequal).

Or, if your language supported finite circular terms (with recursive equality implemented internally via bisimulation), you could recognize all alpha-equivalant terms as equal. I implemented this recently and verified that it works, though I'm not sure it's a good idea. Some pointers on this topic are summarized in http://lists.seas.upenn.edu/pipermail/types-list/2004/000352.html.

Tim Sweeney

Knows his programming language theory too. (Sweeney > Guido van Rossum or Larry Wall.)

The C++/Java/C# Model: "Shared State Concurrency"

- The Idea:
  - Any thread can modify any state at any time.
  - All synchronization is explicit, manual.
  - No compile-time verification of correctness properties:
    - Deadlock-free
    - Race-free

Tim Sweeney

Tim Sweeney's slides from PoPL.

Shared State Concurrency

```
pthread_mutex_lock(mutex);
mutate_shared_variable();
pthread_mutex_unlock(mutex);
```

Lock, mutate/access, unlock, on every access to the variable. Doesn't scale well, is hard to get right (especially if performance is needed: what granularity of locks do you use?). Race conditions, deadlocks, livelocks, no compiler help (though C++: volatile article).

The C++/Java/C# Model: "Shared State Concurrency"

- This is hard!
- How we cope in Unreal Engine 3:
  - 1 main thread responsible for doing all work we can't hope to safely multithread
  - 1 heavyweight rendering thread
  - A pool of 4-6 helper threads
    - Dynamically allocate them to simple tasks.
  - "Program Very Carefully!"
- Huge productivity burden
- Scales poorly to thread counts

Tim Sweeney

...

Concurrency in Gameplay Simulation

This is the hardest problem…

- 10,00's of objects
- Each one contains mutable state
- Each one updated 30 times per second
- Each update touches 5-10 other objects

Manual synchronization (shared state concurrency)
is
hopelessly intractible here.

Solutions?

- Rewrite as referentially-transparent functions?
- Message-passing concurrency?

Tim Sweeney

One of the premiere games programmer and programming language theorists in the entire world is calling this intractible!

A hint of things to come…

The Coming Crisis in Computing

- By 2009, game developers will face…
- CPU's with:
  - 20+ cores
  - 80+ hardware threads
  - >1 TFLOP of computing power
- GPU's with general computing capabilities.
- Game developers will be at the forefront.
- If we are to program these devices productively, you are our only hope!

Tim Sweeney

Don't believe me?

Intel Core Duo: 2 Cores

A laptop. 5 years ago, who'd have thought?

Xbox 360: 3 Cores

Playstation 3: ? Cores

Note: If somebody has a _definitive_ answer for how many cores/SPUs the PS3 has (with sources), please email me!

Sun T2000:
8 Cores, 32 Threads

Available today. Intel Research Labs?

Concurrency in Gameplay Simulation

This is the hardest problem…

- 10,00's of objects
- Each one contains mutable state
- Each one updated 30 times per second
- Each update touches 5-10 other objects

Manual synchronization (shared state concurrency)
is
hopelessly intractible here.

Solutions?

- Rewrite as referentially-transparent functions?
- Message-passing concurrency?

Tim Sweeney

This is the industry-standard approach. How to fix it?

And if you're not convinced by Tim Sweeney…

What's one approach of solving this problem?
Invented at Ericsson, used for telecommunications products.
Should be at 17 minutes.

Erlang the Movie! Teh best movie evar!!

A Functional Language

Hello, World

```
hello() -> io:format( "hello, world!~n" ).

hello( Name ) -> io:format( "hello, ~s!~n", [ Name ] ).
```

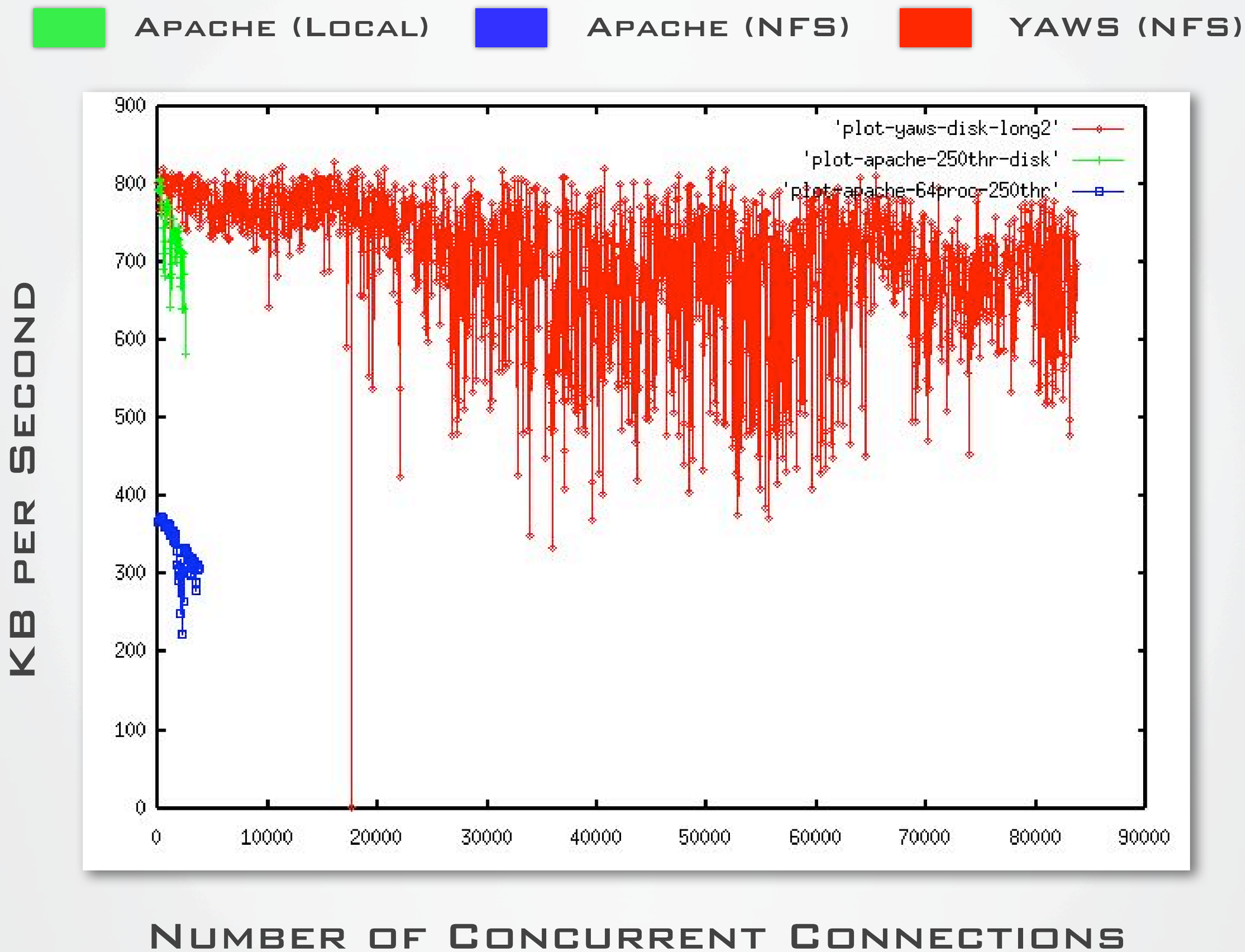
Variable names start with capital letters.
Variable names are single-assignment (const).

Hello, Concurrent World

```
-module( hello_concurrent ).

-export( [ receiver/0, giver/1, start/0 ] ).

receiver() ->
  receive
    diediedie -> ok;
    { name, Name } -> io:format( "hello, ~s~n", [ Name ] ), receiver()
  end.

giver( ReceiverPid ) ->
  ReceiverPid ! { name, "Andre" },
  ReceiverPid ! { name, "SLUG" },
  ReceiverPid ! diediedie.

start() ->
  ReceiverPid = spawn( hello_concurrent, receiver, [] ),
  spawn( hello_concurrent, giver, [ ReceiverPid ] ),
  start_finished.
```

Tuples, spawn used to start new threads, ! used to send messages, and receive used to receive messages. No locking, no mutexes…

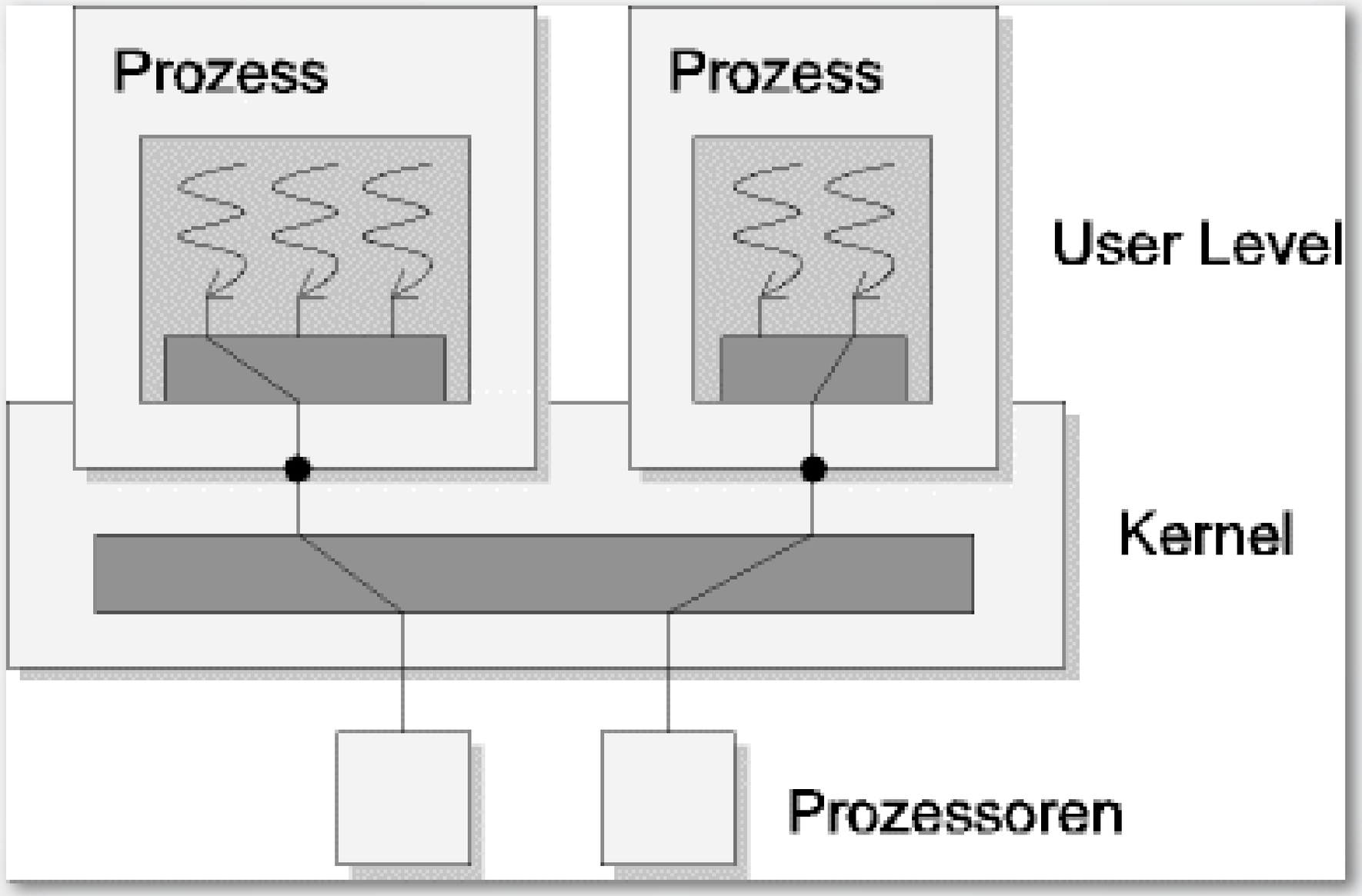
vs



Apache dies at 4,000 connections. YAWS? 80000+…

Why is Erlang so fast?

Speed.



JA JA!

Userland (green) threads. Cooperative scheduler — but safe, because Erlang VM is in full control. Erlang R11B uses multiple kernel threads for I/O and SMP efficiency. No kernel threads means no context switching means very very fast threading. 27 minutes.

Concurrency-oriented Programming?

vs OO vs functional programming.

Each object in the game can be a thread. Why not, when you can have 50000 threads without a problem? Gives you a new approach to thinking about the problem.

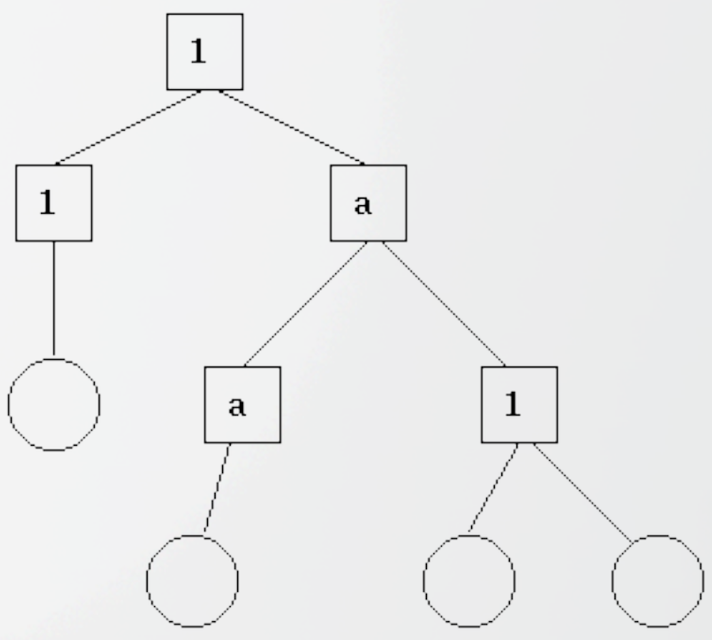
Open Telecommunications Platform (OTP)

Servers

Finite State Machines

Event Handlers

Supervisors



Erlang gives you a complete framework for writing massive, robust, scalable applications. Callback functions. OO analogy. OTP drives the application: you supply the “business logic” as callbacks.

Open Telecommunications Platform (OTP)

Erlang WebTool

WebTool | CrashDumpViewer

Crashdump currently viewed:
/Users/andrep/Crashes/erl_crash-18831.dump

General information
Processes
Ports
ETS tables
Timers
Fun table
Atoms
Distribution information
Loaded modules
Internal Tables
Memory information
Documentation

Load New Crashdump

Process Information Help

Pid	Name/Spawned as	State	Reductions	Stack+heap	MsgQ Length
<0.24235.170>	'ejabberd_mod_logxml_cinesync.com'	Garbing (limited info)	2412669566	145962050	127
<0.23949.170>	erlang:apply/2	Waiting	1138895	2103540	0
<0.24095.243>	proc_lib:init_p/5	Waiting	3923404	832040	0
<0.24983.270>	proc_lib:init_p/5	Scheduled	626994	514229	131
<0.23968.170>	mnesia_tm	Waiting	565816	514229	0
<0.22955.270>	proc_lib:init_p/5	Scheduled	9204702	317811	7
<0.24223.170>	'ejabberd_mod_pubsub_cinesync.com'	Waiting	849347	121393	0
<0.23943.170>	application_master:start_it/4	Waiting	490069	75025	0
<0.23284.243>	proc_lib:init_p/5	Waiting	259456	46368	0
<0.21558.243>	proc_lib:init_p/5	Waiting	252922	46368	0
<0.1590.216>	proc_lib:init_p/5	Waiting	6138384	46368	0

Erlang has good tools required by industry, since it's used in industry as well as academia. e.g. An awesome Crashdump Viewer (or as Conrad would say, Crapdump Viewer).

Open Telecommunications Platform (OTP)

```
erl -rsh /usr/bin/ssh -remsh erlang_node@hostname
1> code:purge(module_name).
2> code:load_file(module_name).
```

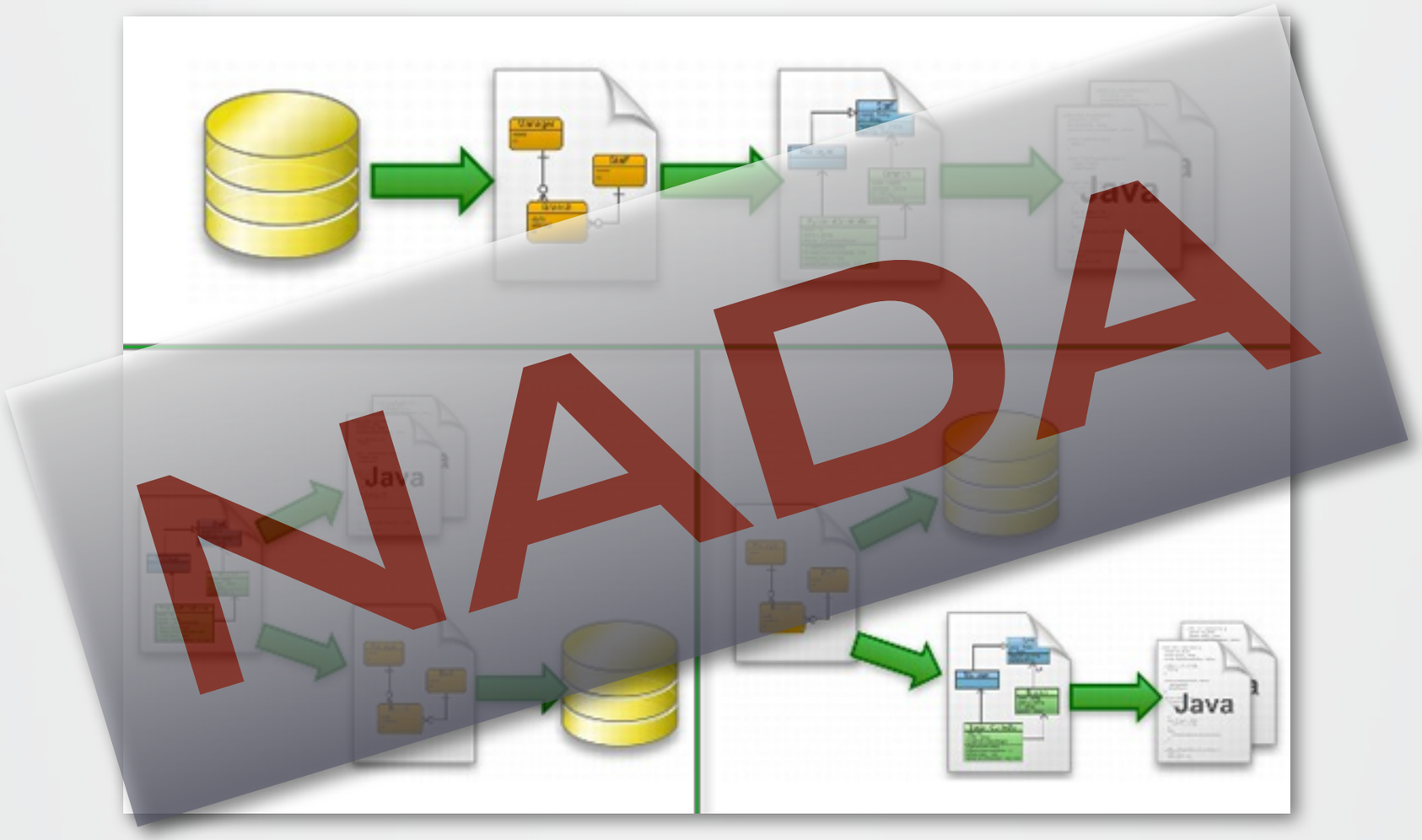
How to do hot-code-reloading: two lines of Erlang! Existing modules will keep running until they're no longer used, all managed by the Erlang VM.

Mnesia

```
-record( passwd, { username, password } ).

mnesia:create_schema( [ node() ] ),
mnesia:start(),
mnesia:create_table( passwd, [] ),
NewUser = #passwd{ username="andrep", password="foobar" },
F = fun() -> mnesia:write( passwd, NewUser ) end,
mnesia:transaction( F ).
```

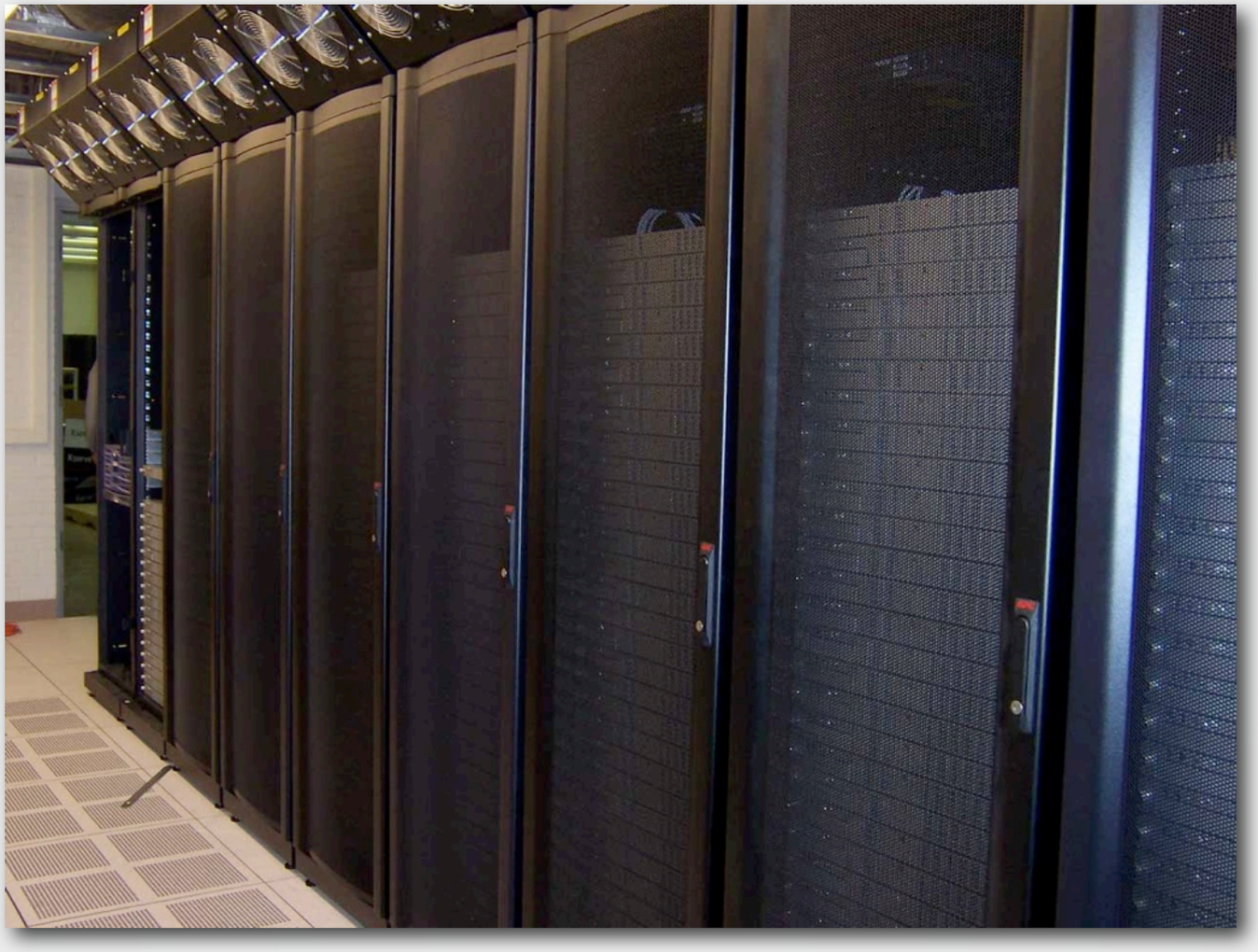
Mnesia is Erlang's insanely great distributed database. Incredibly simple to use!

Mnesia



Object-Relational Mapping. No data impedence mismatch. Store tuples, lists, any Erlang object: none of this SQL row/column nonsense. Query language is just list comprehensions!

Mnesia



Mnesia is replicating. Add new node clusters on-the-fly.

Native Code Compilation

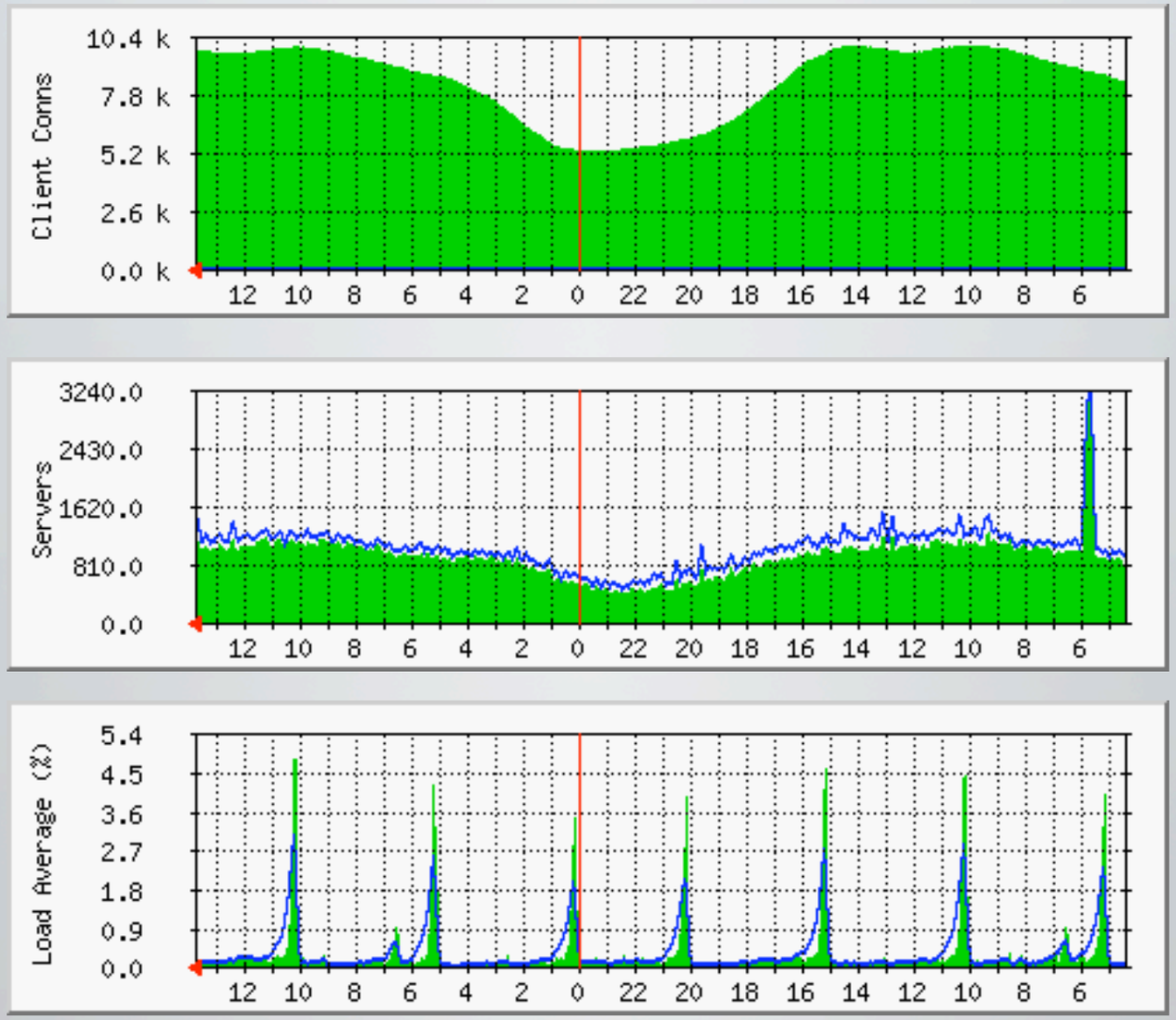
HiPE

Erlang is normally bytecode, but compiles to native code too if you're starting to become CPU-bound.

Robustness

AXD301 telephone switch. One to two million lines of Erlang code. Downtime of maybe a few minutes per year, continuous operation over years. On-the-fly upgrades. Mnesia used for _soft-real-time_ network routing lookup. Mnesia is just 30,000 lines of code. Impressed yet?

Jabber server written in Erlang. High reliability + scalability.
jabber.org, Jabber Australia, Gizmo Project use it.



JABBER.ORG LOAD

5,000-10,000 clients + >800 other Jabber servers all connected to one single machine. Load average is rather low. Also doesn't crash, unlike jabberd2!

=

... WITH EXTRA EXTENSION MODULES.

Why write your own server in C? Why even write your own server in Erlang? Use XMPP as your protocol and use ejabberd instead: distributed, redundant, high availability, high performance application server. Just plug in your own modules for your own business logic.

Questions?

andre.pang@risingsunresearch.com

Thank You!

andre.pang@risingsunresearch.com