

ANDRÉ PANG

# CONCURRENCY AND ERLANG



# THREADS



Concurrency = Threads, for most of you. So, what's so hard about threads?

```
pthread_mutex_lock(mutex);  
mutate_variable();  
pthread_mutex_unlock(mutex);
```

Lock, mutate/access, unlock, on every access to the variable. Looks simple, right?





# WAR STORIES

Well, let's hear some war stories from our own community that may indicate that concurrency isn't quite so easy... (2 minutes)



“A computer is a  
state machine.

Threads are for  
people who can't  
program state  
machines.”

— Alan Cox







Andrew Tridgell: Software Engineering Keynote at Linux.conf.au 2005. In the context of techniques used in Samba 4 to handle multiple concurrent client connections.





“Threads are evil.”





“Processes are ugly...”



“State machines send you mad.”

And this is why Alan Cox’s quip about state machines is, well, slightly incorrect. State machines really do send you mad once you have to handle a metric boatload of states.





“Samba4: Choose your own combination of evil, ugly and mad.”

Similar to Apache: offload the choice to the user. Why does a user have to choose between apache-mpm-event, -itk, -perchild, -threadpool, and -worker threading models? Main point: Tridge is unhappy with all these models.

“... I recall how I took a lock on a data structure that when the system scaled up in size lasted 100 milliseconds too long, which caused backups in queues throughout the system and deadly cascade of drops and message retries...”



“... I can recall how having a common memory library was an endless source of priority inversion problems...”

“... These same issues came up over and over again. Deadlock. Corruption. Priority inversion. Performance problems. Impossibility of new people to understand how the system worked...”



“After a long and careful analysis the results are clear: 11 out of 10 people can't handle threads.”

— Todd Hoff,  
*The Solution to C++ Threading is Erlang*



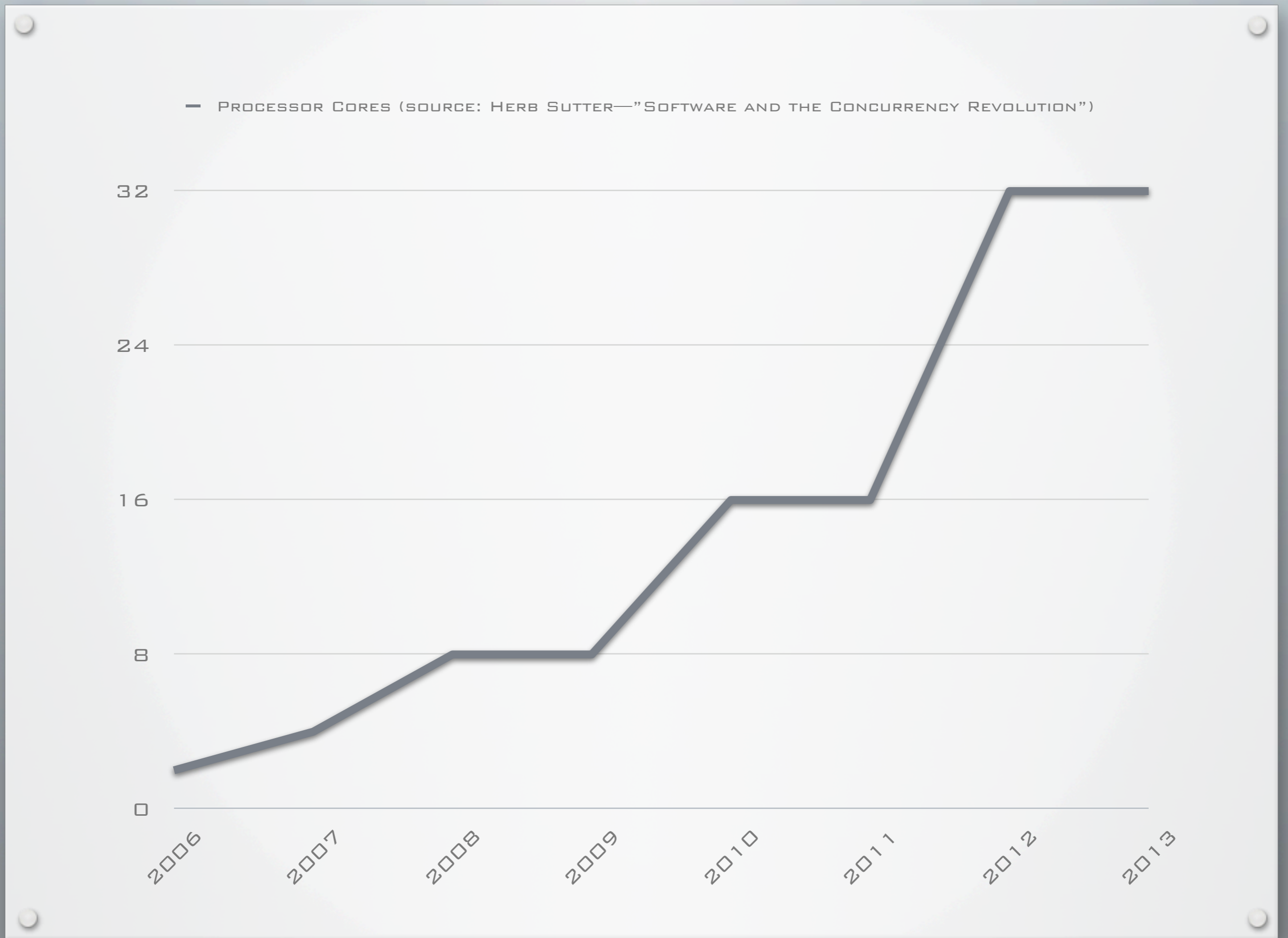
**32 CORES**

Reason 1: Performance, scalability. Servers already have 32 cores. Much bigger challenge to write server code that can scale well to this size. (Apache? Samba?)





**2 CORES**



Reason 2: You'll be required to. Desktops already have 2 cores. Multithreading not important for a lot of applications, but some apps really benefit from them (Photoshop & GIMP!)





Let's talk about an industry that has had to face these problems in the past few years: games!

In the talk, this is was a trailer for Company of Heroes, a state-of-the-art game in 2006 developed by Relic Entertainment. The video was designed to show the interaction of a lot of objects and also the fantastic graphical detail that can be achieved today.





3 Xenon CPUs: PowerPC, 3.2GHz.



©2006 Sony Computer Entertainment Inc. All rights reserved.  
Design and specifications are subject to change without notice.

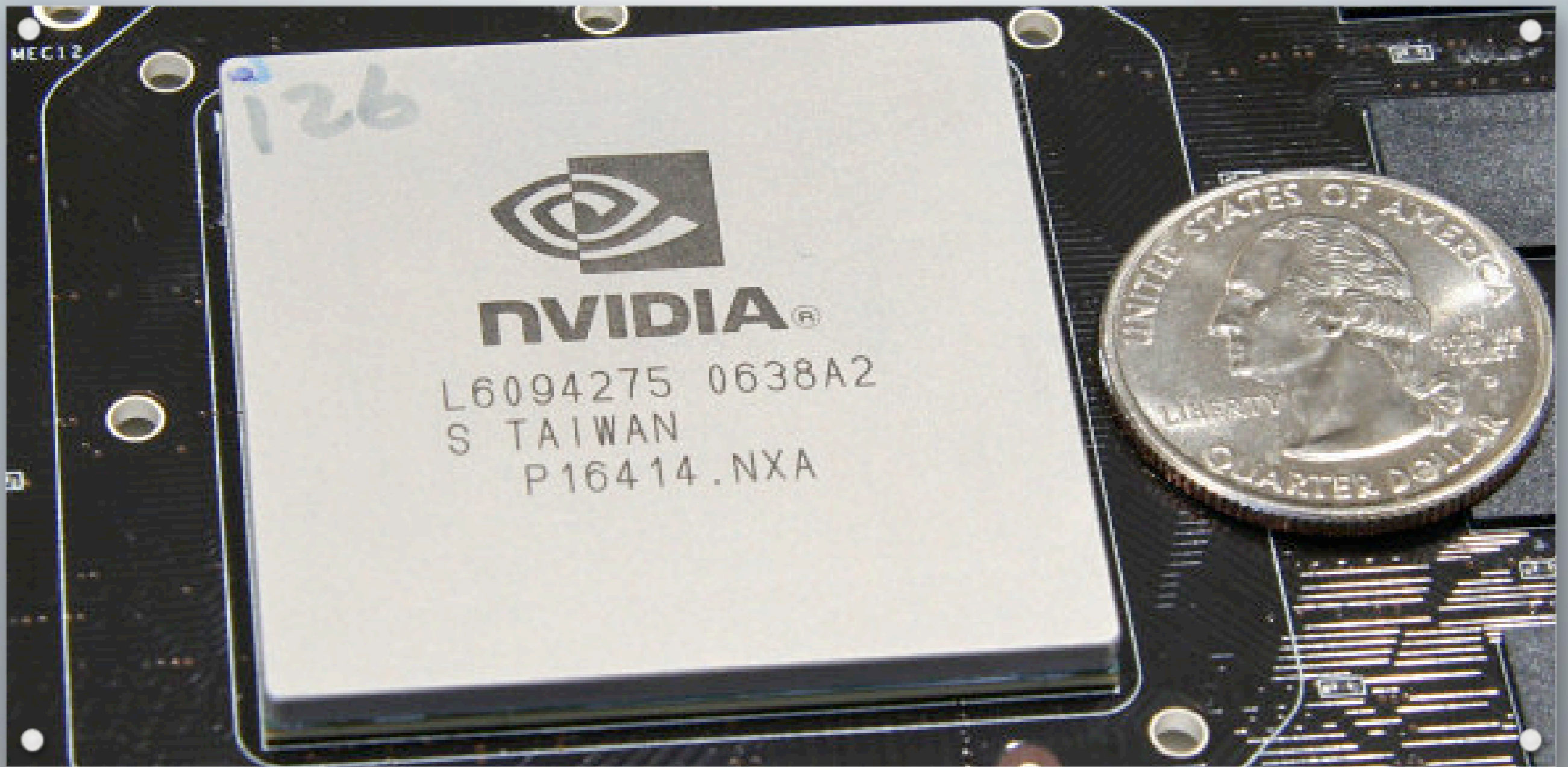
Playstation 3: 1 main PowerPC core @ 3GHz, 6 “Synergistic Processing Elements” at 3GHz.





# GEFORCE 8800

NVIDIA GeForce 8800GTX: 128 stream processors @ 1.3GHz, ~520GFlops.



# GEFORCE 8800

NVIDIA GeForce 8800GTX: 128 stream processors @ 1.3GHz, ~520GFlops.



“If you want to utilize all of that unused performance, it’s going to become more of a risk to you and bring pain and suffering to the programming side.”

— John Carmack





Tim Sweeney: lead programmer & designer, Unreal Tournament (from the original to 2007). Best game engine architect and designer, bar none. Unreal Engine 3 to be sustainable to 2010 (16 cores). 50+ games using UE series. Used in FPSs, RTSs, MMORPGs...



# 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

# 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 intractable here.

Solutions?

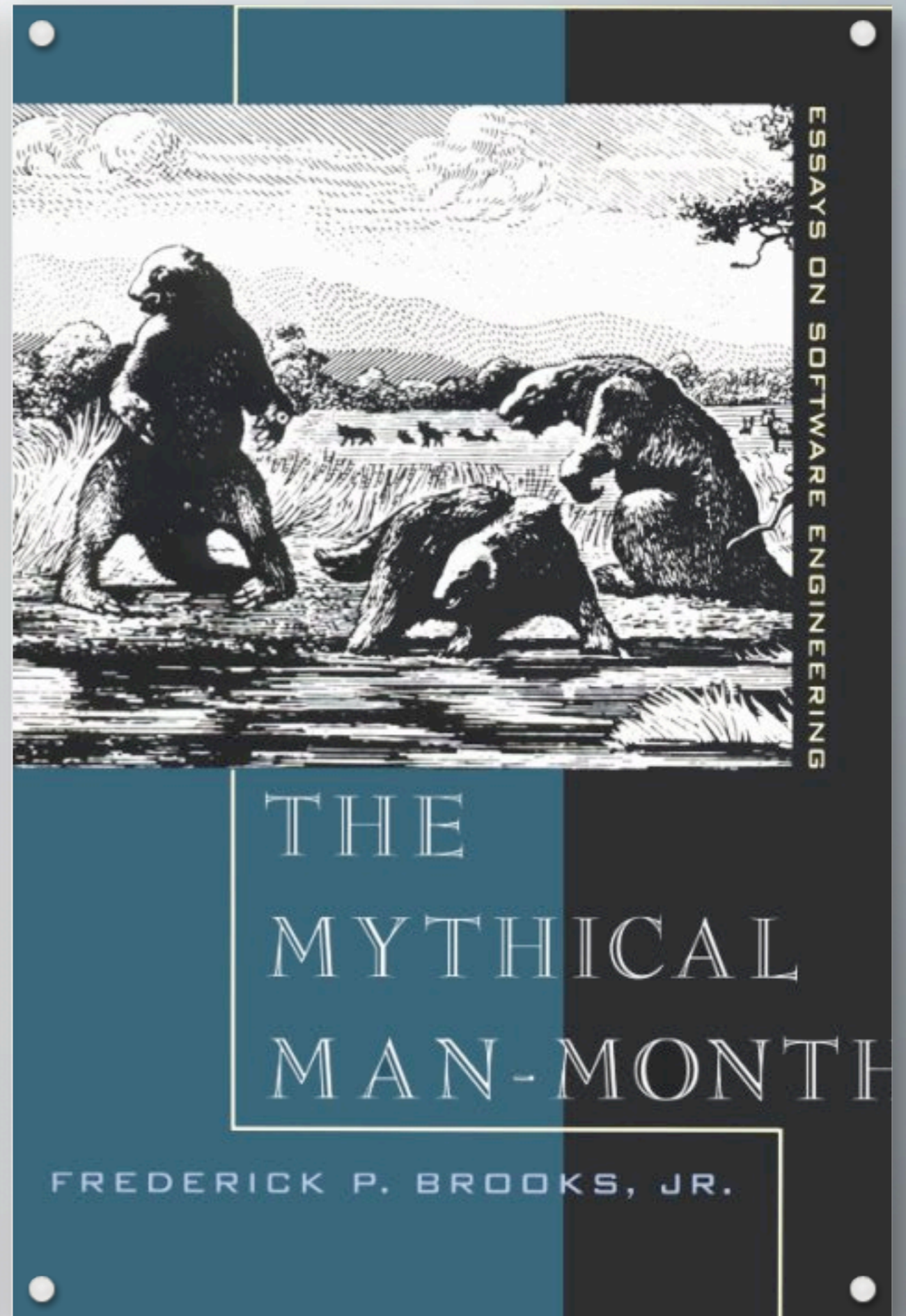


- Rewrite as referentially-transparent functions?
- Message-passing concurrency?



“Surely the most powerful stroke for software productivity, reliability, and simplicity has been the progressive use of high-level languages for programming.”

— Fred P. Brooks





Erlang: a programming language developed at Ericsson for use in their big telecommunications switches. Named after A. K. Erlang, queue theory mathematician. (16 minutes).



# HELLO, WORLD

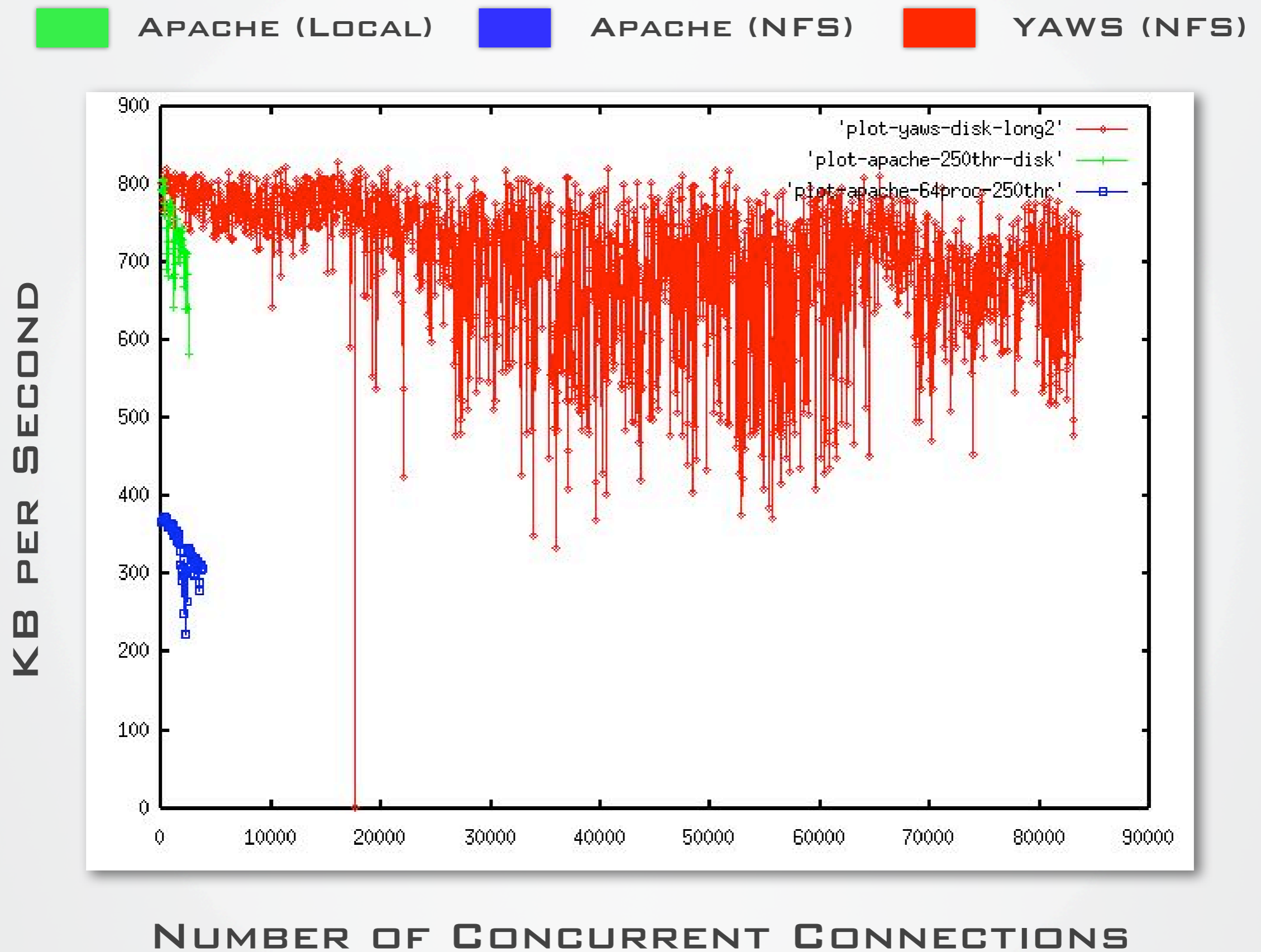
```
hello() -> io:format( "hello, world!~n" ).
```

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

# HELLO, CONCURRENT WORLD

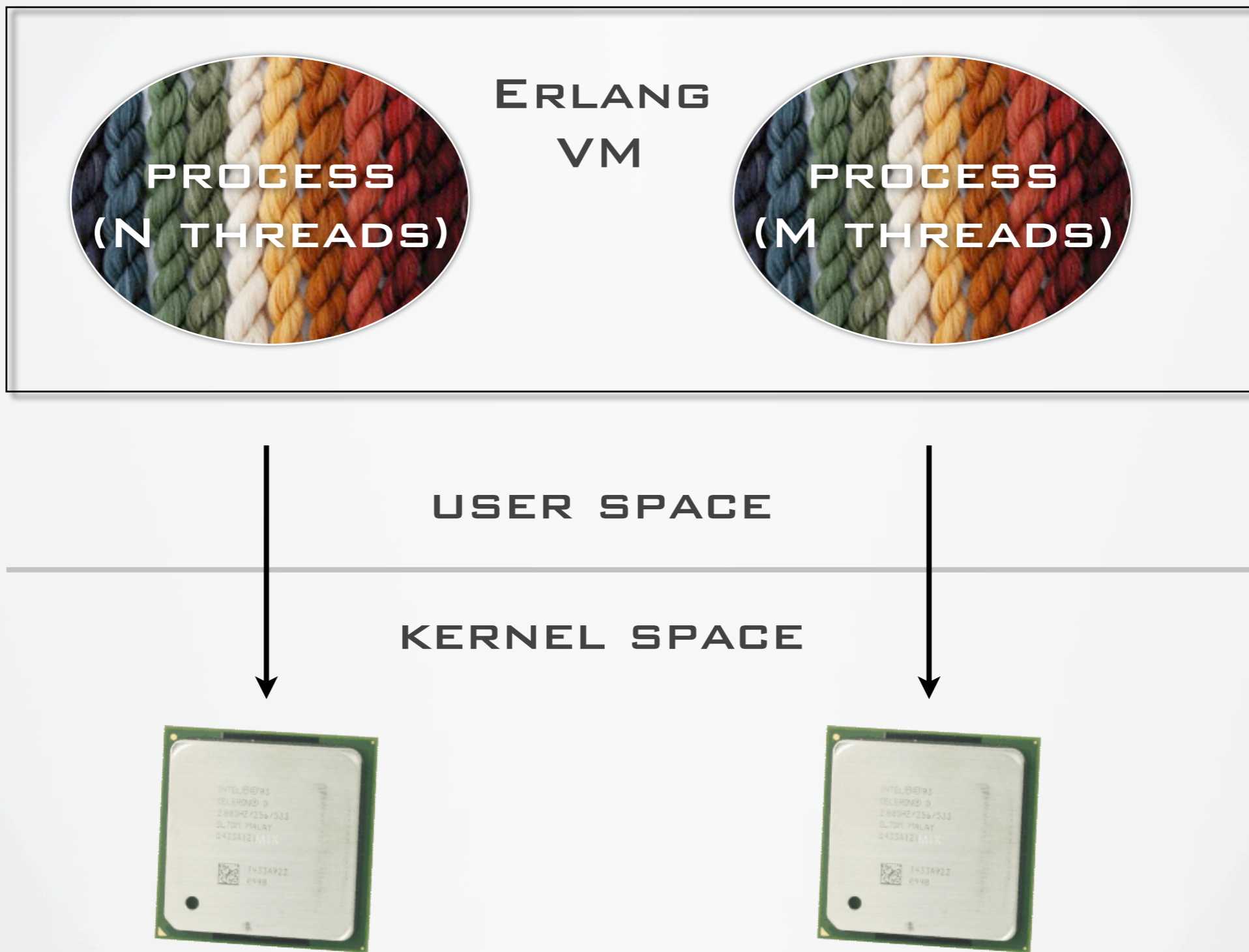
```
-module( hello_concurrent ).  
  
-export( [ receiver/0, giver/1, start/0 ] ).  
  
receiver() ->  
  receive  
    diediedie -> ok;  
    { name, Name } -> io:format( "hello, ~s~n", [ Name ] ), receiver()  
  end.  
  
giver( ReceiverPid ) ->  
  ReceiverPid ! { name, "Andre" },  
  ReceiverPid ! { name, "Linux.conf.au" },  
  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, no shared state at all...



And how is Erlang's performance? Apache dies at 4,000 connections. YAWS? 80000+... (and that's one Erlang process per client connection!)





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.





# WAR STORIES

Flip our problem on its head: what can you do if threads are really easy, instead of being really hard?





# CONCURRENCY- ORIENTED PROGRAMMING

35

Reason 3: Threads can map onto the problem space better. What if every object here was its own thread; its own actor? Wouldn't that be a much more elegant solution than a big gigantic state machine? (25 minutes)



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
<a href="#">&lt;0.24235.170&gt;</a>	'ejabberd_mod_logxml_cinesync.com'	Garbing (limited info)	2412669566	145962050	127
<a href="#">&lt;0.23949.170&gt;</a>	erlang:apply/2	Waiting	1138895	2103540	0
<a href="#">&lt;0.24095.243&gt;</a>	proc_lib:init_p/5	Waiting	3923404	832040	0
<a href="#">&lt;0.24983.270&gt;</a>	proc_lib:init_p/5	Scheduled	626994	514229	131
<a href="#">&lt;0.23968.170&gt;</a>	mnesia_tm	Waiting	565816	514229	0
<a href="#">&lt;0.22955.270&gt;</a>	proc_lib:init_p/5	Scheduled	9204702	317811	7
<a href="#">&lt;0.24223.170&gt;</a>	'ejabberd_mod_pubsub_cinesync.com'	Waiting	849347	121393	0
<a href="#">&lt;0.23943.170&gt;</a>	application_master:start_it/4	Waiting	490069	75025	0
<a href="#">&lt;0.23284.243&gt;</a>	proc_lib:init_p/5	Waiting	259456	46368	0
<a href="#">&lt;0.21558.243&gt;</a>	proc_lib:init_p/5	Waiting	252922	46368	0
<a href="#">&lt;0.1590.216&gt;</a>	proc_lib:init_p/5	Waiting	6138384	46368	0

# CRASHDUMP VIEWER

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 Parker would say, Crapdump Viewer).

# HOT CODE RELOADING

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

# MNESIA

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

```
mnesia:create_schema( [ node() ] ),
```

```
mnesia:start(),
```

```
mnesia:create_table( passwd, [] ),
```

```
NewUser = #passwd{ username="andrep", password="foobar" },
```

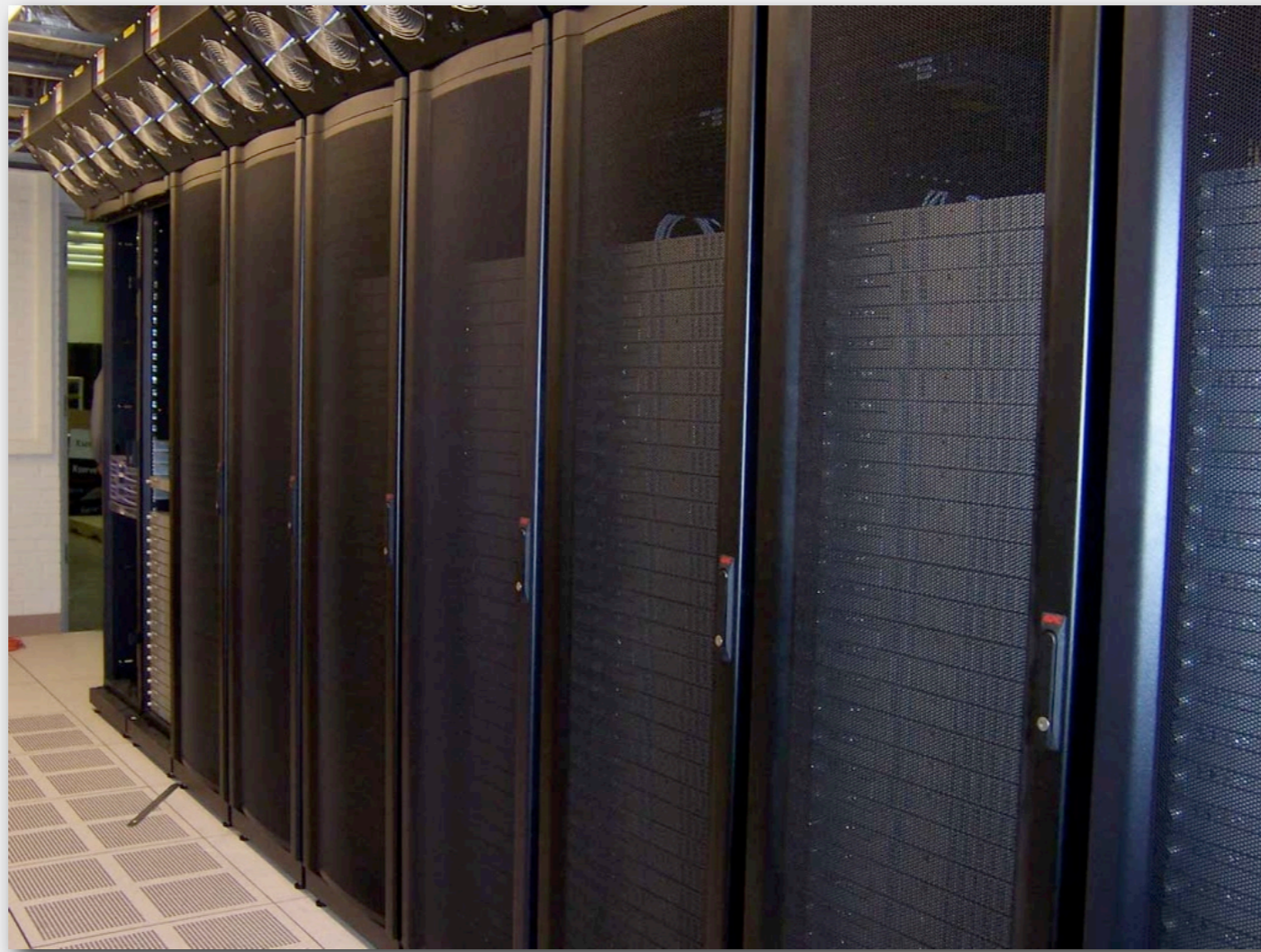
```
F = fun() -> mnesia:write( passwd, NewUser ) end,
```

```
mnesia:transaction( F ).
```

Mnesia is Erlang's insanely great distributed database. Incredibly simple to use! No data impedance mismatch. Store tuples, lists, any Erlang object: none of this SQL row/column nonsense. Query language is just list comprehensions. Transactions are functions!



# MNESIA



Mnesia is replicating. Add new node clusters on-the-fly. Nodes can go down and come back up, and Mnesia will resync the database information to them automatically. With programmer help, it can even recover from network partitioning.

OTP Design Principles

Version 5.5.2

[Bibliography](#) | [Glossary](#) | [Cover](#) | [Up](#)

Table of Contents

1 [Overview](#)

1.1 [Supervision Trees](#)

1.2 [Behaviours](#)

1.3 [Applications](#)

1.4 [Releases](#)

1.5 [Release Handling](#)

2 [Gen\\_Server Behaviour](#)

2.1 [Client-Server Principles](#)

2.2 [Example](#)

2.3 [Starting a Gen\\_Server](#)

2.4 [Synchronous Requests - Call](#)

2.5 [Asynchronous Requests - Cast](#)

2.6 [Stopping](#)

2.7 [Handling Other Messages](#)

3 [Gen\\_Fsm Behaviour](#)

3.1 [Finite State Machines](#)

3.2 [Example](#)

3.3 [Starting a Gen\\_Fsm](#)

3.4 [Notifying About Events](#)

3.5 [Timeouts](#)

3.6 [All State Events](#)

3.7 [Stopping](#)

3.8 [Handling Other Messages](#)

4 [Gen\\_Event Behaviour](#)

4.1 [Event Handling Principles](#)

4.2 [Example](#)

4.3 [Starting an Event Manager](#)

4.4 [Adding an Event Handler](#)

4.5 [Notifying About Events](#)

4.6 [Deleting an Event Handler](#)

4.7 [Stopping](#)

The OTP Design Principles is a set of principles for how to structure Erlang code in terms of processes, modules and directories.

### 1.1 Supervision Trees

A basic concept in Erlang/OTP is the **supervision tree**. This is a process structuring model based on the idea of **workers** and **supervisors**.

- Workers are processes which perform computations, that is, they do the actual work.
- Supervisors are processes which monitor the behaviour of workers. A supervisor can restart a worker if something goes wrong.
- The supervision tree is a hierarchical arrangement of code into supervisors and workers, making it possible to design and program fault-tolerant software.

*Supervision Tree*

In the figure above, square boxes represent supervisors and circles represent workers.

### 1.2 Behaviours

In a supervision tree, many of the processes have similar structures, they follow similar patterns. For example, the supervisors are very similar in structure. The only difference between them is which child processes they supervise. Also, many of the workers are servers in a server-client relation, finite state machines, or event handlers such as error loggers.

# SELF-HEALING ARCHITECTURE

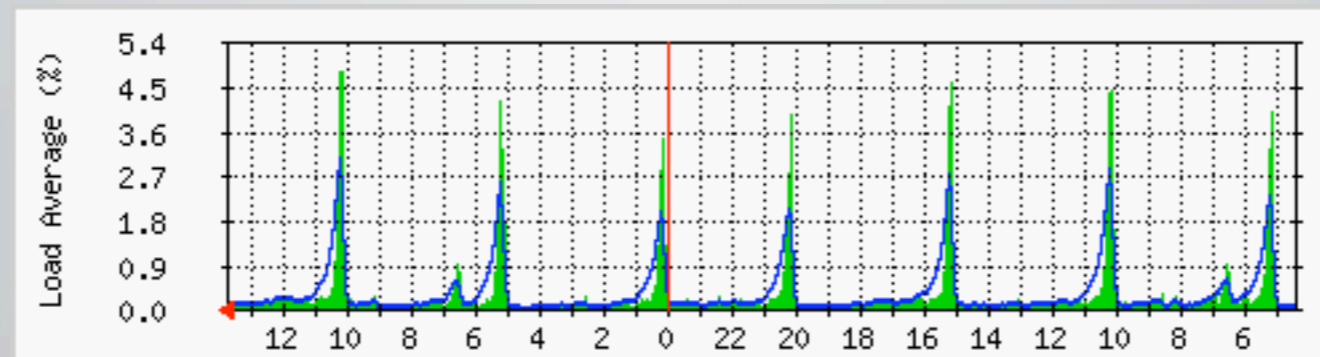
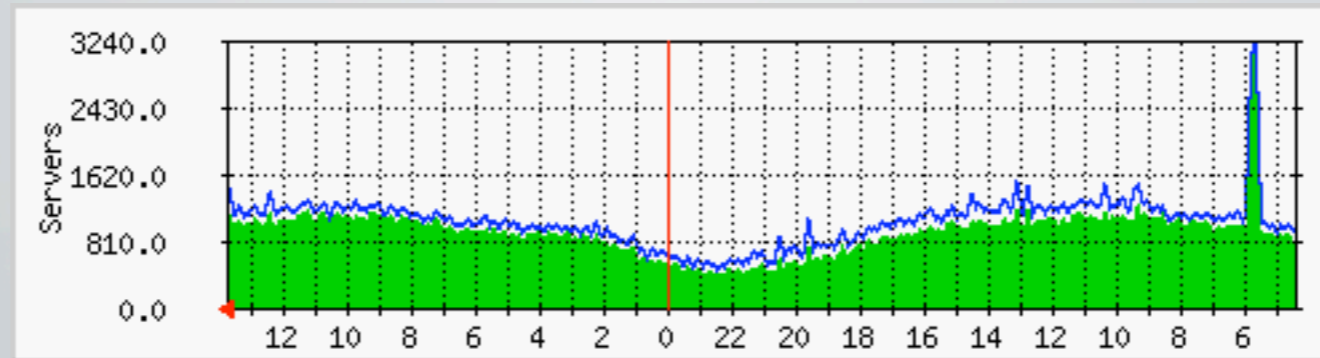
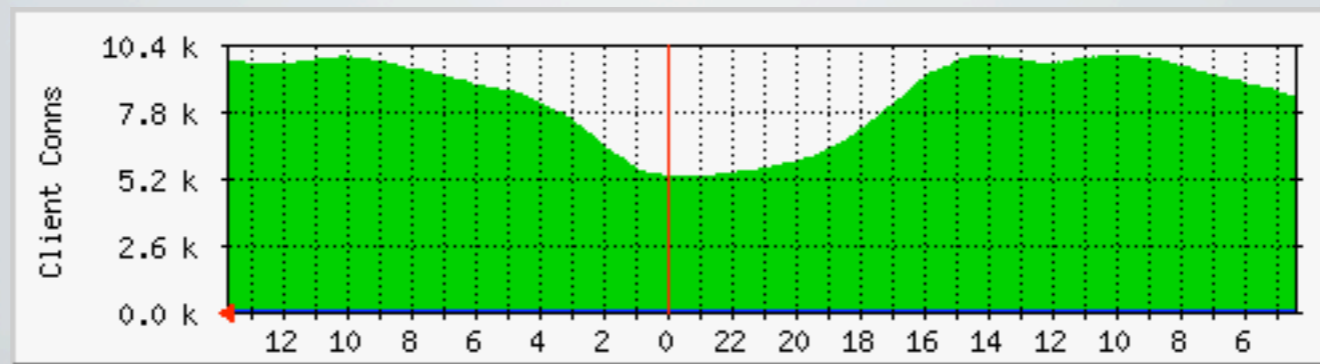
Erlang gives you a complete framework for writing massive, robust, scalable applications. Callback functions. OO analogy. OTP drives the application: you supply the “business logic” as callbacks. This means that Erlang is a self-healing, self-sustaining system, and is the main reason why Erlang applications are so robust. (30 minutes)





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?





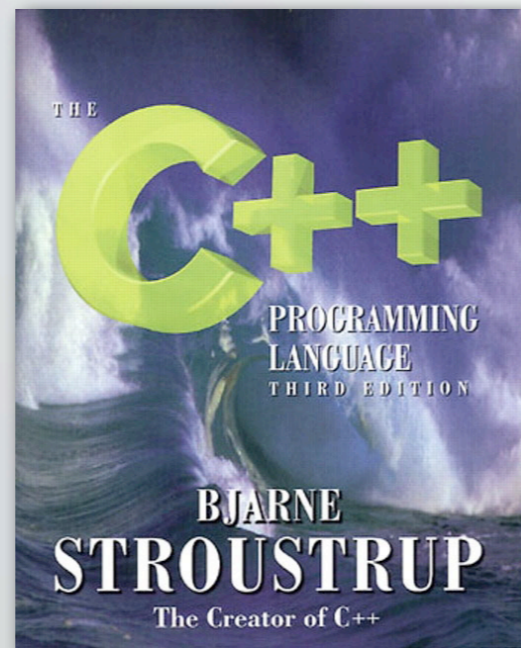
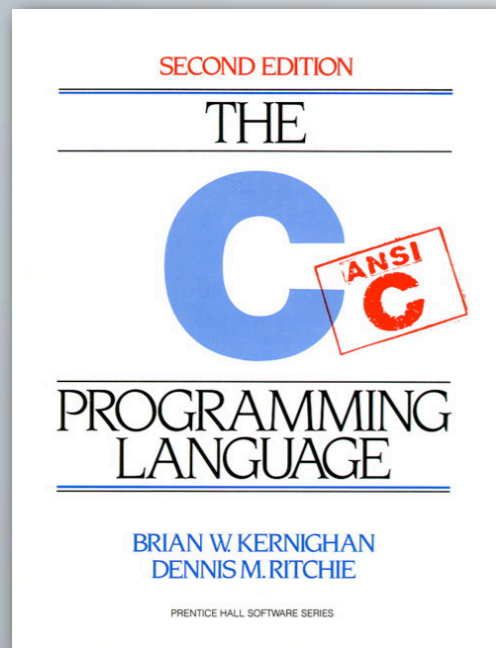
5,000–10,000 clients + >800 other Jabber servers all connected to one single machine. Load average is rather low. Also doesn't crash, unlike jabberd2!

```
pthread_mutex_lock(mutex);  
mutate_variable();  
pthread_mutex_unlock(mutex);
```

**JUST SAY NO**

Shared state concurrency just 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. Just say no!





```
        j < KC; ++j)
    for(i = 0; i < 4; ++i)
        tk[i][j] ^= tk[i][j - 1];

for(j = 1; j < KC / 2; ++j)
    for(i = 0; i < 4; ++i)
        tk[i][j] ^= tk[i][j - 1];

for(i = 0; i < 4; ++i)
    tk[i][KC / 2] ^= boxes.getElementAt(
        KC / 2 + 1; j < KC; ++j)
```



```
    new Account[] {
        ...
    }

/** Add a data line to the internal data container for
 *  <code>planeFields</code> indicate
 *  found on the current line, which may be less than t
 *  The remaining members of plane will be empty <code>t
 *  @param plane data fields
 *  @param planeFields number of actual data fields
 */
public void addLine(String[] plane, int planeFields) {
    if (planeFields > MAXIMUM_FIELDS) {
        planeFields = MAXIMUM_FIELDS;
    }
}
```

Prefer the messaging (actor) model: use it in your own language! You can do it in C, C++, Python, Java, or whatever your other language is. You may have to write some infrastructure code, but by God it'll be easier in the end!



# QUESTIONS?



[ANDRE.PANG@RISINGSUNRESEARCH.COM](mailto:ANDRE.PANG@RISINGSUNRESEARCH.COM)

# THANK YOU!



ANDRE.PANG@RISINGSUNRESEARCH.COM