Erlang

Productivity and Performance

My Perspective

- Using Erlang for 3+ years on industrial projects

- Amazon for 5 years
  - working on tier-1 stateful distributed systems

- Valve LLC for 3 years
  - did most of the core backend for www.steampowered.com (~20 million registered users, 1.88 million concurrent users)
  - in C++, which drove me to look for Erlang

- Before that: designed/wrote video-games

Overview

- Introduction To Erlang
- Productivity
- Performance
- Erlang on Multi-Core

Introduction To Erlang

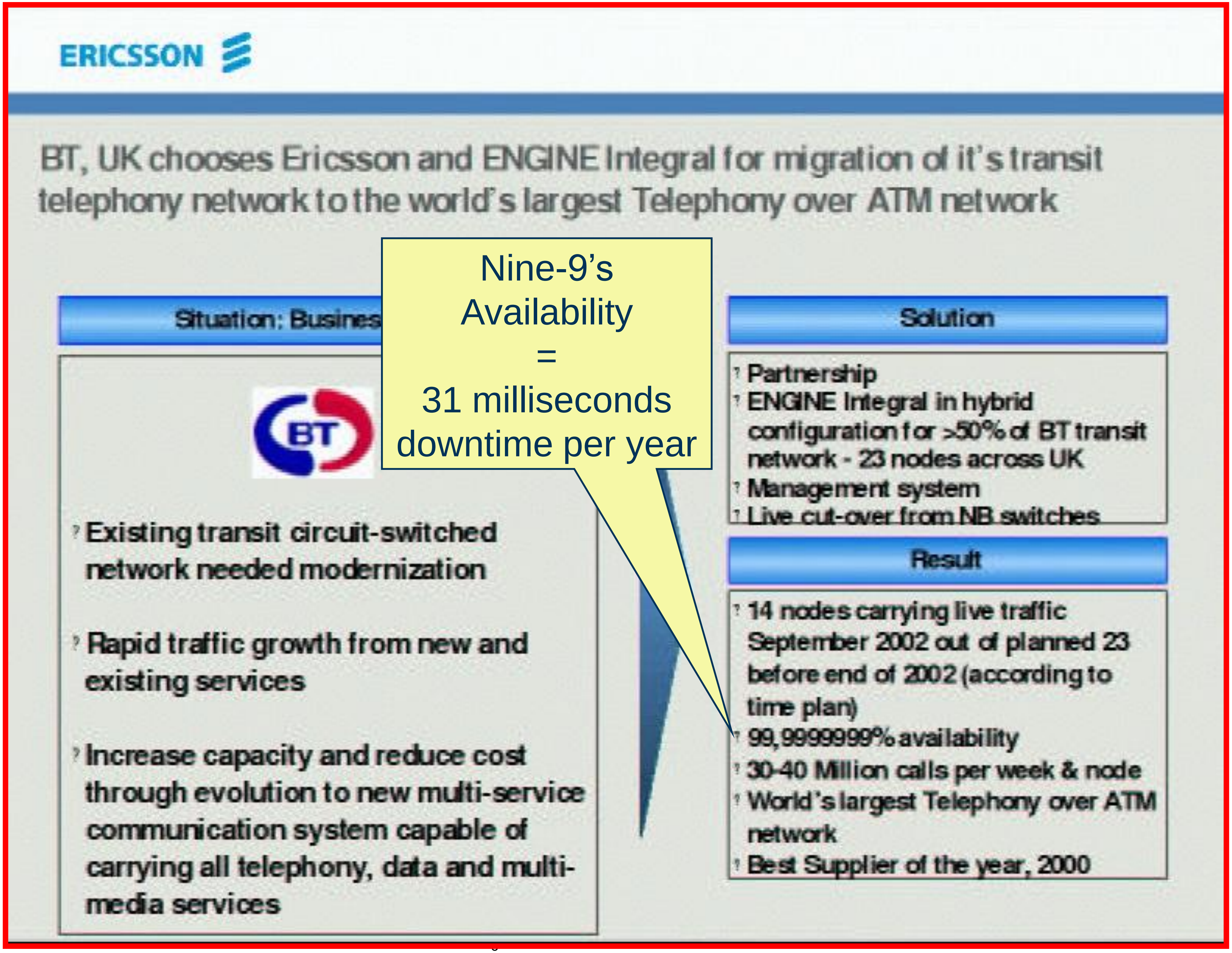
- Motivation
- The Big Idea
- Primary Mechanisms

Motivation for Erlang

- Make it easier to build extremely robust, high-end telecoms switches

- Biggest availability issue is software defects

- Biggest productivity issue is complexity of concurrent interactions
  - large nested state machines
  - usual distributed-system issues
    (e.g. power-set of partial-failure modes)



ERICSSON

BT, UK chooses Ericsson and ENGINE Integral for migration of it's transit telephony network to the world's largest Telephony over ATM network

Nine-9's
Availability
=
31 milliseconds
downtime per year

Situation: Busines

- Existing transit circuit-switched network needed modernization
- Rapid traffic growth from new and existing services
- Increase capacity and reduce cost through evolution to new multi-service communication system capable of carrying all telephony, data and multi-media services

Solution

- Partnership
- ENGINE Integral in hybrid configuration for >50% of BT transit network - 23 nodes across UK
- Management system
- Live cut-over from NB switches

Result

- 14 nodes carrying live traffic September 2002 out of planned 23 before end of 2002 (according to time plan)
- 99,9999999% availability
- 30-40 Million calls per week & node
- World's largest Telephony over ATM network
- Best Supplier of the year, 2000

The Big Idea
A New Internal Architecture

- Applications composed of **isolated, loosely-coupled micro-services, communicating via asynchronous message-passing**

- Fault-tolerance via “supervisors”:
  - micro-services that monitor and restart other micro-services
  - hierarchical escalating restart (recovery-oriented computing)

- **Micro-services and message-passing should be so cheap that they become the default abstraction**
  - thousands of ‘active objects’ / ‘actors’

- Linear control-flow, even when doing IO in thousands of processes
  - VM implements scheduler, hides details of async. IO

- Avoid features that break robustness and distribution
  - mutable memory-shared state, conventional mutexes, synchronous interaction between processes

Primary Mechanisms

- Many isolated ‘erlang processes’
  - one-to-one concurrency with problem domain
  - reasonable to have hundreds-of-thousands of processes
  - VM is a single OS process, perhaps one OS thread per core
- Processes are kill-safe and crash-safe
  - fail-fast error handling
- Processes can monitor each other and receive an asynchronous signal or a message when another process exits
- Each process has a private mailbox
  - message-delivery does not interrupt receiver process
  - default FIFO ordering
  - can ‘selectively receive’ (consume out of order) via pattern-matching

Language Overview

- Syntax inspired by prolog
    - but different semantics (simple linear control-flow)
- Pervasive pattern-matching
- Small set of types; atom, number, list, tuple, binary, closure
- Strong, dynamic (runtime) type-checking
- No explicit pointers/references
- Immutable data values, possibly sharing internal structure
    - pure-functional algorithms required for data-structures
- Bind-once variables (via pattern-matching)
    - no assignment operator
- No conventional OO support
    - but processes are ‘true’ objects (see Alan Kay’s OOPSLA 97 keynote)
- Constant-space tail-calls
    - Looping done with recursion or high-order functions, as in Scheme

Language Overview continued

- ‘Mutable state’ provided by subsystems with ‘service API’
  - copy data on both read and write
  - ETS, Mnesia, Berkeley DB, …
- Sophisticated runtime tracing features
- Live code loading/replacement
- Some cruft
  - broken lexical scope
  - flat module namespace
  - relatively poor/expensive string handling
  - rather ad-hoc libraries
  - awkward conditional control-flow (if/case)
  - performance issues (see later)
- Open-source, superbly maintained by Ericsson
  - no external committer rights
- Other flavors
  - LFE (Lisp Flavored Erlang)
  - Reia (“script language”, allows rebinding of variables)

Quick Overview of Erlang Syntax

```
-module(math).
-export([fac/1]).

fac(N) when N > 0 ->  N * fac(N-1);
fac(0)            ->  1.

> math:fac(25).
15511210043330985984000000
```

Append

```
% append([1,2,3], [4,5]) = [1,2,3,4,5]
%
% Same as List1 ++ List2
% (copies List1, shares structure with List2)

append([H | T], List2) ->
   [H | append(T, List2)];

append([], List2) ->
   List2.
```

Binary Search Tree

```
% A node is  {Key, Value, LeftSubtree, RightSubtree}
%         or  nil

lookup(Key, {Key, Val, _, _}) ->
   {ok, Val};

lookup(Key, {NodeKey, Val, L, R}) when Key < NodeKey ->
   lookup(Key, L);

lookup(Key, {NodeKey, Val, L, R}) ->
   lookup(Key, R);

lookup(Key, nil) ->
   not_found.
```

High-Order Functions / Closures

```
> Adder = fun(Increment) ->
             fun(N) -> N + Increment end
          end.
#Fun

> G = Adder(10).
#Fun

> G(7).
17
```

Concurrency

```
 % Create a process

Pid = spawn(fun() ->
                do(),
                things()
            end).

% Send a message to a process

Pid ! {my_msg, With, [“Arbitrary”, Structure]}.
```

Selectively receive a message

```
% All receive-patterns are tested against first message
% in mailbox, then against second message, and so on.

receive
   {my_msg, _, [FirstElem, _]} ->
       % some actions (presumably using FirstElem);
  … snip any number of patterns/actions …
   AnyMsg ->
       % more actions
after
   TimeoutMillisecs ->
       % … actions
end.
```

Create and monitor a process

```
% Choose to convert async. ‘exit’ signals to messages
% (only supervisors/coordinators should do this)

process_flag(trap_exit, true),

% ‘links’ are bi-directional
% (there is a uni-directional variant)

Pid = spawn_link(fun() -> ... end),

receive
    {’EXIT’, Pid, Reason} ->
            % actions ...
end
```

“Behaviors”

- Remove the boilerplate from common patterns

gen_server	basic micro-service
gen_event	simple publish/subscribe
gen_fsm	convenient state machines
supervisor	monitor and restart other processes

gen_leader	process pool with leader election
plain_fsm	allows nested state machines

- Good overview doc. : OTP Design Principles

Other Patterns

- GProc : Extended Process Registry
  - “find the right process”
  - indexed meta-data for processes, with automatic cleanup
  - ‘references/pointers’ in a loosely-coupled world
- Other Ulf Wiger code
- ERESYE Erlang Expert System Engine and Linda-style tuple-space
- Erlang Questions mailing list archives

Productivity

Productivity
For which problem-domain?

- Erlang is excellent for **industrial-scale systems** with certain goals
  - Fault-tolerant
  - Soft real-time
  - Highly concurrent
  - Distributed (from wire-level protocols to high-level choreography)

- Currently poor for
  - Intensive numerical computation
  - Mutation-heavy computation
  - Most micro-benchmarks

Dimensions of Productivity

- Expressivity of syntax
- Expressivity of abstractions
- Convenience of error-handling, resource management
- Breadth and quality of library support
- Ease of interfacing to libraries in other languages
- Reliability, maturity
- Support for debugging
- Support for maintenance of existing code/systems
- Support for operations of running systems
- Performance (how much optimization is required?)

Dimensions of Productivity

- Expressivity of syntax
  - pattern-matching is great
  - ‘bit-syntax’ and ‘binaries’ are great for implementing low-level protocols
- Expressivity of abstractions
  - processes, message-passing, links are a huge win
  - can directly model the concurrency of the problem-domain
  - avoids ‘gimbal lock’ of conventional shared-memory concurrency
- Convenience of error-handling, resource management
  - Good exception support (try/catch/after)
  - BUT hard-killing a process bypasses any catch/after clauses
    - other processes should monitor and do clean-up
  - Any ‘ports’ owned by the process (e.g. sockets, files) *are* always closed when it exits
    - mechanism is painful to customize - requires C code

Dimensions of Productivity cont.

- Breadth and quality of libraries
  - good for telecoms, otherwise relatively ad-hoc / poor. Improving slowly.
- Ease of interfacing to libraries in other languages
  - somewhat painful painful
  - philosophy is good: treat all external entities as processes; send/receive messages and assume they may crash
  - have to wrap APIs in message-passing interface
- Reliability/maturity
  - world class
- Support for debugging
  - excellent : trace facilities, remote shells, visibility tools
- Support for maintenance of existing code/systems
  - excellent : hot code-loading, clean concentration of state for ‘upgrade’
- Support for operations of running systems
  - excellent : remote shells, visibility tools
- Performance (how much optimization is required?)

More Information

- "Four-fold increase in productivity and quality" (2001) http://citeseer.ist.psu.edu/wiger01fourfold.html
- “Concurrency Oriented Programming In Erlang” http://www.sics.se/~joe/talks/ll2_2002.pdf
- “Erlang Rationale” http://www.trapexit.org/forum/viewtopic.php?p=44172
- "History of Erlang" http://www.cs.chalmers.se/Cs/Grundutb/Kurser/ppxt/HT2007/general/languages/armstrong-erlang_history.pdf
- "World-class product certification using Erlang" (2002) http://citeseer.ist.psu.edu/old/wiger02worldclass.html
- "Troubleshooting a large Erlang system" (2004) http://www.erlang.se/workshop/2004/cronqvist.pdf
- "Verification of Distributed Erlang Programs using Testing, Model Checking and Theorem Proving“ http://www.cs.chalmers.se/~hanssv/doc/PhDThesis.pdf
- "AXD 301 A new generation ATM switching system" (1998) http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.5674

Performance

Current Performance Issues

- Dynamic (runtime) type checking
- Immutable data-values
  - O(1) factors become O(lg N) and generate garbage
- “Public” mutable state is copied on both read and write
  - and any sharing of sub-structure is lost
- Byte-code based VM, relatively few compiler optimizations
  - constant factors are relatively high compared to C, Java
  - native-code compiler improves things but is rarely used
- Copy on send, and any sharing of sub-structure is lost
- “Message-passing API” to third-party low-level libraries, may incur marshalling / copying

Performance Strengths

- Garbage-collection is per-process (and generational)
  - root-set and live-set are usually tiny
  - likely to be fine-grain, non-blocking
- Transient processes with pre-sized heaps can often avoid g.c. entirely
- Large binary data is reference-counted
- ETS is not scanned by garbage collector at all

- See Erlang Efficiency Guide

Support Material

Industry Case Study

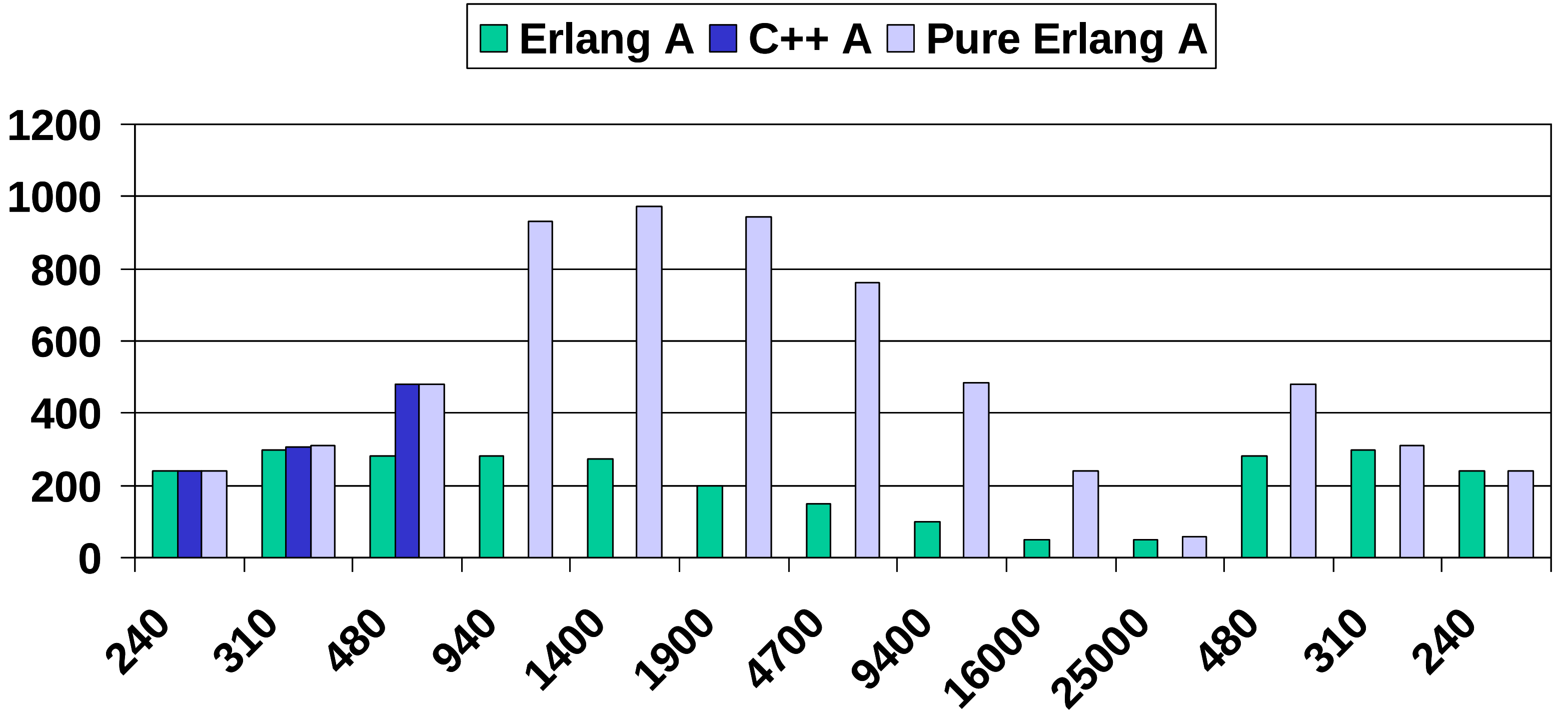
A research team worked with Motorola Telecoms to re-implement two existing C++ components of a production mobile-phone system in pure Erlang, and a mixture of Erlang/C.

http://www.erlang.se/euc/06/proceedings/1600Nystrom.ppt

Erlang vs C++
Motorola Telecoms

- Code size:
  1. Erlang version **1/7 the size of the C++** original (398 lines vs. 3101),
  2. Erlang version **1/3 the size of the C++** original (4,882 lines vs. 14,900)
- Throughput
  - Erlang version **2x throughput of the existing C++ version** (before QoS started to degrade in both versions)
- Latency
  - Erlang version **3x faster (roundtrip times) than the C++ version**
- Availability
  - Erlang version available throughout repeated induced hardware failures
  - No data for C++ version
- Resilience
  - Erlang version **never failed** even at **overload of 25,000 requests per second**.
  - C++ version failed before reaching 1,000 requests per second.

Reslience



X axis is load (queries per second)

Y axis is throughput (queries per second)