Internship report

Edouard Swiac

November 2011

Acknowledgments

I would like to thanks Dr. Leeds, supervisor.

Abstract

Most *Voice over IP* service providers create home-grown tools to manage their systems. As their needs, and their customer's demands, grow, their toolset falls behind and becomes a pain point for their company. A system built to grow with millions of users in mind must be conceived to be scalable from start, and that scalability can be achieved by relying on a distributed architecture. Complexity is abstracted in the Cloud, therefore hidden from the end-user.

Let enter VoIP to the Cloud.

Contents

Chapter 1

Introduction

Chapter 2

Using Erlang/OTP and telephony systems

functional, concurrent programming language, write multicore programs to scale fault-tolerant applications that can be modified without taking them out of service functional programming lamnguage that has been battke tested in real large scale industruak products, with great librairies and an active use community virtues of functional programming functional programming forbis code with side effects. side effects and concurrency don't mix, You can have sequential code with side effects, or you can have code and concurrency that is free from side effects. no middle way Erlang is a language where concurrency belongs to the programming language and not the operating system. Erlang makes parallel program- ming easy by modeling the world as sets of parallel processes that can interact only by exchanging messages. In the Erlang world, there are parallel processes but no locks, no synchronized methods, and no pos- sibility of shared memory corruption, since there is no shared memory. Erlang programs can be made from thousands to millions of extremely lightweight processes that can run on a single processor, can run on a multicore processor, or can run on a network of processors. It's about concurrency. It's about distribution. It's about fault toler- ance. It's about functional programming. It's about programming a dis- tributed concurrent system without locks and mutexes but using only pure message passing. It's about speeding up your programs on multi- core CPUs. It's about writing distributed applications that allow people to interact with each other. It's about design methods and behaviors for writing fault-tolerant and distributed systems. It's about modeling concurrency and mapping those models onto com-

puter programs, a process I call concurrency-oriented programming. OTP a high-level, concurrent, robust, soft real-time system that will scale in line with demand T-Mobile uses Erlang in its SMS and authentication systems. Motorola is using Erlang in call processing products in the public-safety industry. Ericsson uses Erlang in its support nodes, used in GPRS and 3G mobile networks worldwide. mid-1980s, Ericsson's Computer Science Laboratory was given the task of in- vestigating programming languages suitable for programming the next generation of telecom products. Erlang was influenced by functional languages such as ML and Miranda, concurrent languages such as ADA, Modula, and Chill, as well as the Prolog logic programming language. Erlang was developed to solve the "time-to-market" requirements of distributed, fault-tolerant, massively concurrent, soft real-time systems. The fact that web services, retail and commercial banking, com- puter telephony, messaging systems, and enterprise integration, to mention but a few, happen to share the same requirements as telecom systems explains why Erlang is gaining headway in these sectors. Concurrency in Erlang is fundamental to its success. Rather than providing threads that share memory, each Erlang process executes in its own memory space and owns its own heap and stack. Processes can't interfere with each other inadvertently, as is all too easy in threading models, leading to deadlocks and other horrors. Processes communicate with each other via message passing, Message passing is asynchronous, Erlang concurrency is fast and scalable. Even though Erlang is a high-level language, Because of this, Erlang can handle high loads with no degradation in throughput, even during sustained peaks. OTP is simultaneously a framework, a set of libraries, and a methodology for structuring applications; it's really a language extension

These are some of the main advan- tages of OTP: Productivity—Using OTP makes it possible to produce production-quality sys- tems in a very short time. Stability—Code written on top of OTP can focus on the logic and avoid error- prone reimplementations of the typical things that every real-world system needs: process management, servers, state machines, and so on. Supervision—The application structure provided by the framework makes it sim- ple to supervise and control the running systems, both automatically and through graphical user interfaces. Upgradability—The framework provides patterns for handling code upgrades in a systematic way. Reliable code base—The code for the OTP framework is rock solid and has been thoroughly battle tested. — the core concepts and features of the Erlang/OTP platform that everything else in OTP builds on: Concurrent programming -

Fault tolerance - Distributed programming - The Erlang virtual machine and runtime system - Erlang's core functional language

Chapter 3

Agile development and software development lifecycle in a startup environment

startup = few people, few money, great project agile refcard software development and professional practice

Chapter 4

Getting knowledge of the software lifecycle and the management of an Open Source project

software management and professional practices opensource bouquin jaune vend support , code open source

Chapter 5

Understanding how VoIP, Cloud and distributed systems are related

blog 2600hz We get a lot of questions about why we selected Erlang and FreeSWITCH for our new switching platform. Most engineers have never used it. The best way to conquer misinformation is to provide better quality details with supporting facts.

Who Uses Erlang?

Erlang is often misunderstood as old, slow, or little used. Actually, Erlang powers some of the technology behind Amazon, Facebook and even your IP Video set top box that your cable company runs. The simplest explanation behind Erlang's power lies in it's history. About 20 years ago, Joe Armstrong created the Erlang language while trying to solve the problem of distributed computing for telecom. The basic premise for developing Erlang? It was believed that at some point it would be impossible to make a computer faster and redundant. While the rest of the world wanted faster processors, faster memory, faster hard disks, Joe and his team believed that "fast" would hit a wall one day. Instead, he believed getting computers to work together, as a collection of resources, made much more sense. Joe's philosophy for Erlang is illustrated in a recounting he shared on the Erlang mailing list:

"Once upon a time I was talking to a guy and used the word 'fault tolerance.' We talked for hours and thought we understood what this word meant. Then he said something that implied he was building a fault-tolerant system on one computer. I said it was impossible. He said, 'You catch and handle

all the exceptions . . . ' I said, 'The entire computer might crash.' He said, 'Oh.' There was a long silence. I told him about Erlang . . . "

Little did that team know they were developing a technology that would be in high demand many years later. Today, Google is the most well-known master of such distributed technology, with a platform that utilizes cheap commodity hardware to store massive amounts of data in a redundant, distributed manner. Google spreads the work it has to do across many computers to scale their service. This way of running computers is now generally referred to as "scaling horizontally." The older way of designing software to run on a single or a few really fast computers is known as "scaling vertically."

What We Needed

The requirements to build a scalable, distributed platform aren't a mystery to anyone who's built a distributed system. Even Google has a tutorial on how to do so (http://code.google.com/edu/parallel/dsd-tutorial.html). But most people's experience in distributed systems revolve around web or database platforms. Distributed VoIP on commodity platforms is still a reasonably new phenomenon (we'll get into why in a separate post) and it has different requirements.

Generally Speaking, a Scalable, Distributed platform generally consists of these components:

* Messaging: A way for programs across different servers to talk to each other and/or know what other servers are doing

* Redundancy: The ability to have copies of everything (data, software, etc.) all working together at the same time, with copies coming online/offline at any given time

* Distribution of Data: The ability to break information into pieces and spread it across multiple computers, allowing for adding/removing computers as demand requires

* Unlimited Concurrency as a Concept: The idea that there should be no limit to how much is happening on the platform overall

The above are common characteristics. But VoIP is unique in how it works and has additional requirements. Our needs also included:

* Directed Events: The ability for message queues across boxes to be spun up and down quickly and to act as a "tunnel" between different explicit services without disrupting other nodes

* Schema Flexibility: The ability to frequently upgrade data and variable structures within the entire system without bringing down clusters (inherently having different versions of schema running while the system, as a

whole, remains operational)

* Strong Supervision: The ability to detect failures *very* quickly and re-spawn nodes and processes just as fast. In web servers, delays and failures of 50ms or more are acceptable - in voice applications, they're ultimately not

* Speed for Adding Features: Telecom is growing extremely rapidly. The ability to expose new features quickly, in a reliable, scalable, distributed way is paramount to a successful platform

* Fast Server Provisioning: The ability to handle spikes "in the cloud" by procuring and provisioning additional resources (from servers to circuits to DIDs) in an instant

* The ability to move in-progress calls around to servers that have better network connectivity or lower latency

* The ability to avoid ALL downtime (as a goal) - not even upgrades to software should result in downtime.

Selecting Erlang

We spent a lot of time looking for all the pieces above. Foolishly, we were looking for them individually. We thought of using beanstalk and Java or even Amazon's hosted service for messaging. All of them would require setting up our own queues and brokering systems. Some cost money. We thought about using PHP and lighthtppd / nginx / Apache for the web portions. We thought about using the event socket and something (we never did figure that out) for the real-time streaming. We played with Comet extensively for the browser component and tried out XML and JSON.

But all in all, everything felt cobbled together and there were always large gaps.

We didn't just read about these technologies, either - we've actually tried many different languages (Perl, Python, PHP, Ruby) as well as databases (Postgres, MySQL, SQLite, MongoDB, CouchDB). Only through trial and error have we really been able to assess the strengths and weaknesses of each platform through use of specific VoIP applications - not just general research and theory.

When we found Erlang: It literally was like heaven when we began exploring it's capabilities. It excelled in every single item we listed above out-of-the-box and in the last three years has seen much expansion on it's few weaknesses. For example, the built-in Mnesia database system was really not acceptable for our purposes and required too much manual labor to maintain, but CouchDB (also written in Erlang) would fill that gap. RabbitMQ filled any gaps in messaging, and FreeSWITCH already had an Erlang connector.

It was a no brainer to dive into this technology, and it has served us very well in both reliability and scalability.

The Results

We will soon begin posting statistics and information about our findings with Erlang in great depth, but in the meantime, let's just say that it's exceeded all expectations.

After researching many languages and options, we basically ended up with two choices. Option one was to code our platform in C, making it more popular and easier to tie in with other code libraries but likely to require months of work designing a highly scalable distributed threading and messaging system ourselves (or integrate with one like ZeroMQ) along with threading, supervision and parsing engines. Option two was to use a programming language where someone had already done all that work to help give us a head-start, in the hopes that the language itself would increase in popularity as others began facing similar problems as listed above.

We chose the latter. Since doing so, Erlang's popularity has grown, and we're convinced it was the right choice to get us moving in the right direction.

Here's what Erlang gave us for free:

*Powerful Messaging

* Powerful Data Storage

* Native types and connections between messaging and data storage engines

* Ability to Swap Code on Running Systems (0 downtime as a goal)

* Ability to Distribute Compiled Code Cross-Platform

* Massively scalable processes/threading

We'll dive in more into our architecture and Erlang code in upcoming blogs. Stay tuned!

Chapter 6

How to build a scalable and distributed VoIP platform

http://blog.2600hz.com/post/5399315067/erlang-and-freeswitch-the-future-of-cloud maybe merging both

Chapter 7

What you have learned/achieved

learned erlang developed module on whistle managed tickets and priorities working with other people, =/= cow boy coder testing code

Chapter 8

How you believe this experience will impact your future/carriere

open source technologies important in the silicon valley learning tools used by the community

Chapter 9

Conclusions

Bibliography

[1] 2600hz company's blog. http://blog.2600hz.com/.

[2] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Swedish Institute of Computer Science, 2003.

[3] Joe Armstrong. *Programming Erlang*. The Pragmatic Programmers, LLC, 2007.

[4] Francesco Cesarini and Simon Thompson. *Erlang Programming*. O'Reilly Media, Inc., 2009.

[5] John Dooley. *Software Development and Professional Practice*. Apress Editions, 2011.

[6] Martin Logan, Eric Merritt, and Richard Carlsson. *Erlang and OTP in Action*. Manning Publications Co., 2011.

[7] Google Code University. Introduction to distributed system design http://code.google.com/edu/parallel/dsd-tutorial.html.