Certificate of Management and Marketing of IT

Internship report

Author:
Edouard Swiac

Supervisor:
Dr. Christopher Leeds

September 5, 2011

Acknowledgments

I would like to thanks Dr. Christopher Leeds, my supervisor and business communication teacher, and the Dominican University of California for having me as a foreign student in its certificate program.

An internship begins by being accepted in a company. 2600hz gave me a lot, from a personal and a professional perspective. James Aimonetti and Karl Anderson were supportive and taught me. Without them this experience would not have been so enriching.

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.

Contents

Chapter 1

Context of the internship

1.1 Company

1.2 Product

The company's main software, Whistle, is designed to handle up to a billion calls per month on about six virtualized (or not) servers and can connect seamlessly to run on or with Rackspace clouds, Amazon's clouds or on a private cluster of machines. Instead of paying a penny or so per minute to a VoIP company, businesses that want to add voice calling over the web to their social network, their app or their role-playing game just deploy this software and take care of it themselves.

Whistle, the company's next-generation communications engine's core platform was built for scale. Your media servers, configuration databases and call handling logic can all be spread across the internet in any way you see fit. Fully utilize all that the Cloud has to offer. Its entire management and configuration capabilities are exposed via simple APIs. Did we mention, it's free? Welcome to the next generation of VoIP.

The plan is to offer folks the source code for Whistle, but to support and sell services on top of it, in much the same that every other open source software company plans to make money. Schreiber says the services include a voice mail platform, a minute reseller platform, a conference call serve and will include many more.

1.3 About the internship

The intern joined a team of 2 senior software engineers, who each have 10+ years of experience in programming and telephony systems architecture. The role of the intern was to contribute to the development of Whistle, developing Erlang module and being fully integrated in the Whistle team.

Chapter 2

Erlang/OTP in telephony systems

2.1 Overview of Erlang

Erlang is a general-purpose concurrent, garbage-collected programming language and runtime system. The sequential subset of Erlang is a functional language, with strict evaluation, single assignment, and dynamic typing.

It was designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications. It supports hot swapping, so that code can be changed without stopping a system. Hot swapping is particularly suitable for high availability systems like telephony systems.

Erlang was developed to solve the "time-to-market" requirements of distributed, fault-tolerant, massively concurrent, soft real-time systems, and was designed with the aim of improving the development of telephony applications.

The language has been battle tested in real large scale industrial products, like the AXD301 switch in 1998, containing over a million lines of Erlang, and reported to achieve a reliability of 99.999% (5.26 minutes downtime / year).

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.

2.2 Strengh of Erlang for telephony system

One of the greatest strength of Erlang for scalable soft real-time communication systems is its concurrency model. Erlang concurrency is fast and scalable, and can handle high loads with no degradation in throughput, even during sustained peaks. Telephony systems must not suffer from call peaks to be reliable.

Erlang makes parallel programming easy by modeling the world as sets of parallel processes that can interact only by exchanging asymchronous messages. 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. There are no locks, no synchronized methods, and no possibility of shared memory corruption, since there is no shared memory, which ensure a high stability of the platform, hence lowered downtimes.

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. Concurrency in Erlang belongs to the programming language and not the operating system, which makes Erlang portable over different platforms (UNIX/Linux, Windows, Mac OS) and architures (x64, x86).

Functional programming forbids code with side effects, which Erlang implements as referential transparency. Referential transparency requires the same results for a given set of inputs at any point in time thus a referentially transparent expression is therefore deterministic by definition. Bugs and errors in a deterministic system are easier to reproduce than in a nondetermistic one.

2.3 Open Telephony Platform

OTP is simultaneously a framework, a set of libraries, and a methodology for structuring applications; it's an extension of Erlang. These are some of the main advantages of OTP:

Productivity Using OTP makes it possible to produce production-quality systems 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.

2.4 Applications of Erlang/OTP in industry-level communication systems

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.

Facebook is using Erlang in the back-end of its chat system.

The fact that web services, retail and commercial banking, computer 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.

Chapter 3

Agile development

3.1 Introduction to Agile practices

The vast majority of software projects suffer from a steady degradation of design quality and it becomes more and more difficult to maintain the software with the same level of quality. As the software ages it calcifies and becomes harder and harder to maintain. In some cases it becomes too expensive to maintain and so it the software is put to rest and rewritten. In others, the software is released with a steadily increasing number of defects. Both of these common situations are deeply unsatisfying, but there are several practices that can be set up to reduce defects, improve design and increase quality.

Many of the practices from the Agile world stop the degradation of software quality and turn the trend around. It is not unheard of for teams to have maintained a zero-defect status for months and years. Design and architecture have become malleable; they now emerge and transform over time.

Below are some practices used by the company the student worked for.

3.2 Refactoring

The practice of Refactoring code changes the structure (i.e., the design) of the code while maintaining its behavior. Incremental improvement of design is the name of the game with refactoring; continuous refactoring keeps the design from degrading over time, ensuring that the code is easy to understand,

maintain, and change.

3.3 Continuous Integration

Continuous integration reduces the defects in a software system by catching errors early and often and enabling a stop-and-fix process. It leverages both automated acceptance tests and automated developer tests to give frequent feedback to the team and prompts removing these defects promptly.

3.4 Collective Code Ownership

Collective code ownership means that members of a development team have the right and responsibility to modify any part of the code. They get more exposure to the entire code base and are able to remove defects wherever they are found and incrementally modify the design of the system accordingly.

3.5 Evolutionary Design

Evolutionary design is the simple design practice (below) done continuously. Teams start off with a simple design and change that design only when a new requirement cannot be met by the existing design.

3.6 Simple Design

If a decision between coding a design for today's requirements and a general design to accommodate for tomorrow's requirements needs to be made, the former is a simple design. Simple design meets the requirements for the current iteration and no more.

3.7 Iteration

An iteration is a time-box where the team builds what is on the backlog and is a potential release and therefore enables building less and forces regularly removing defects to reach the agreed upon done state.

3.8 Release Often

Releasing your software to your end customers as often as you can without inconveniencing them forces you to constantly have your software in releasable quality and allows you to build in smaller increments and get feedback before too much of an investment is made.

3.9 Stand Up Meeting

Stand up meetings are daily meetings for the team to synch-up and share progress and impediments daily. This helps keep the entire team aware of what is being done and where in the system.

Chapter 4

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

5.1 Distributed Platform

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

5.2 Specifications of a VoIP platform

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.

Chapter 6

Distributed VoIP platform design

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

6.1 Why 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 ex-

ploring 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.

6.2 Platform design

-OpenSIPs -FreeSWITCH -Whistle -RabbitMQ -WhApps

wh-msg ecallmgr

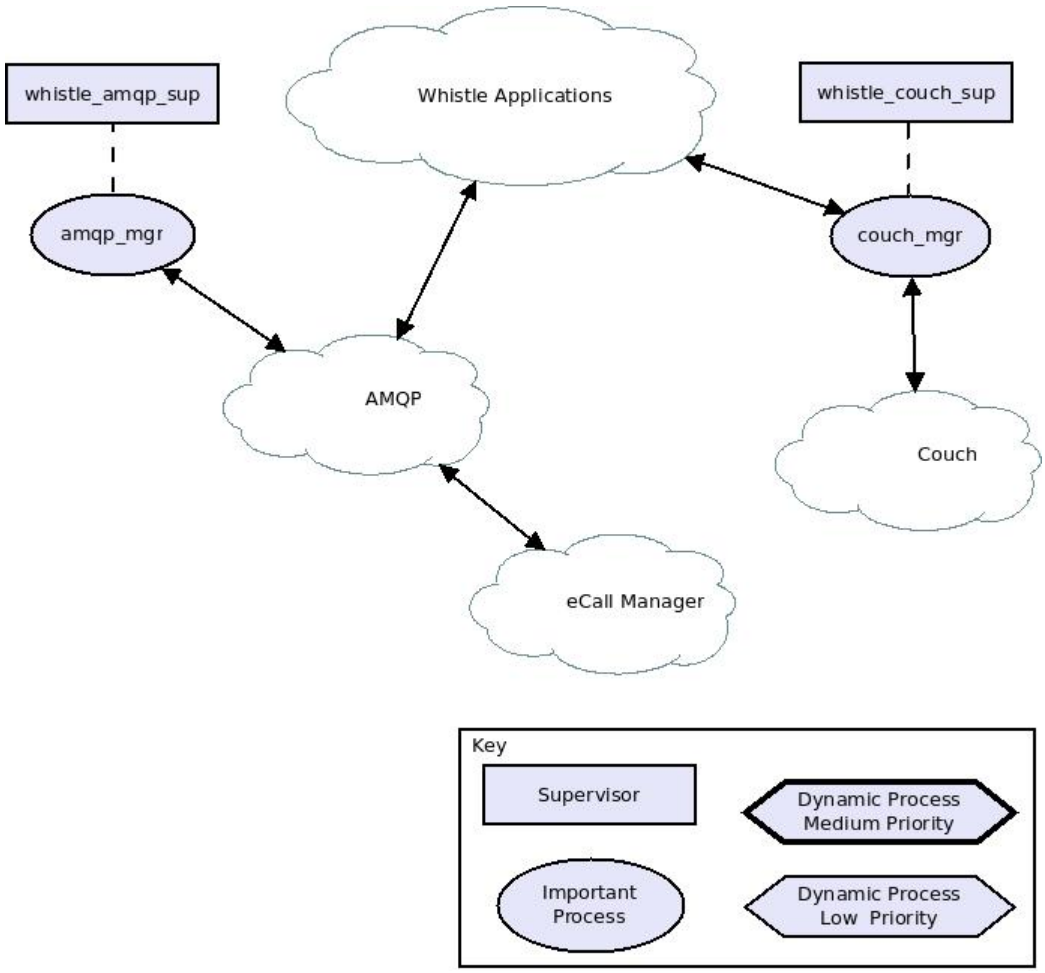
We will soon begin posting statistics and information about our findings with Erlang in great depth, but in the meantime, let's just say that it's exceeded all expectations.

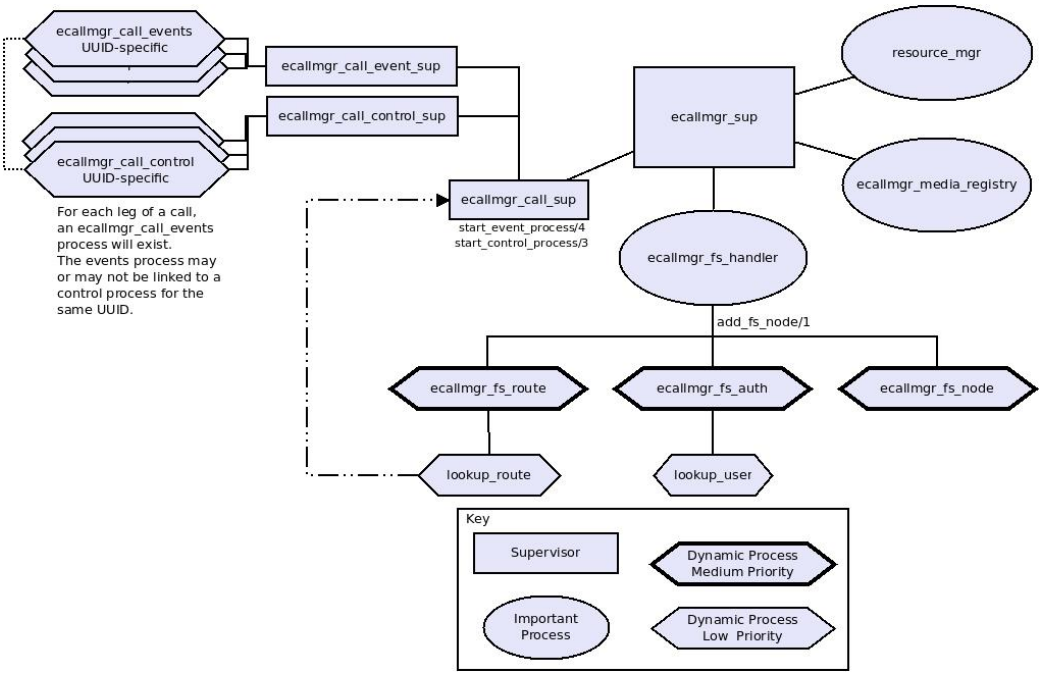
After researching many languages and options, we basically ended up with two choices. Option one was to code our platform in C, making it more popular and easier to tie in with other code libraries but likely to require months of work designing a highly scalable distributed threading and messaging system 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.

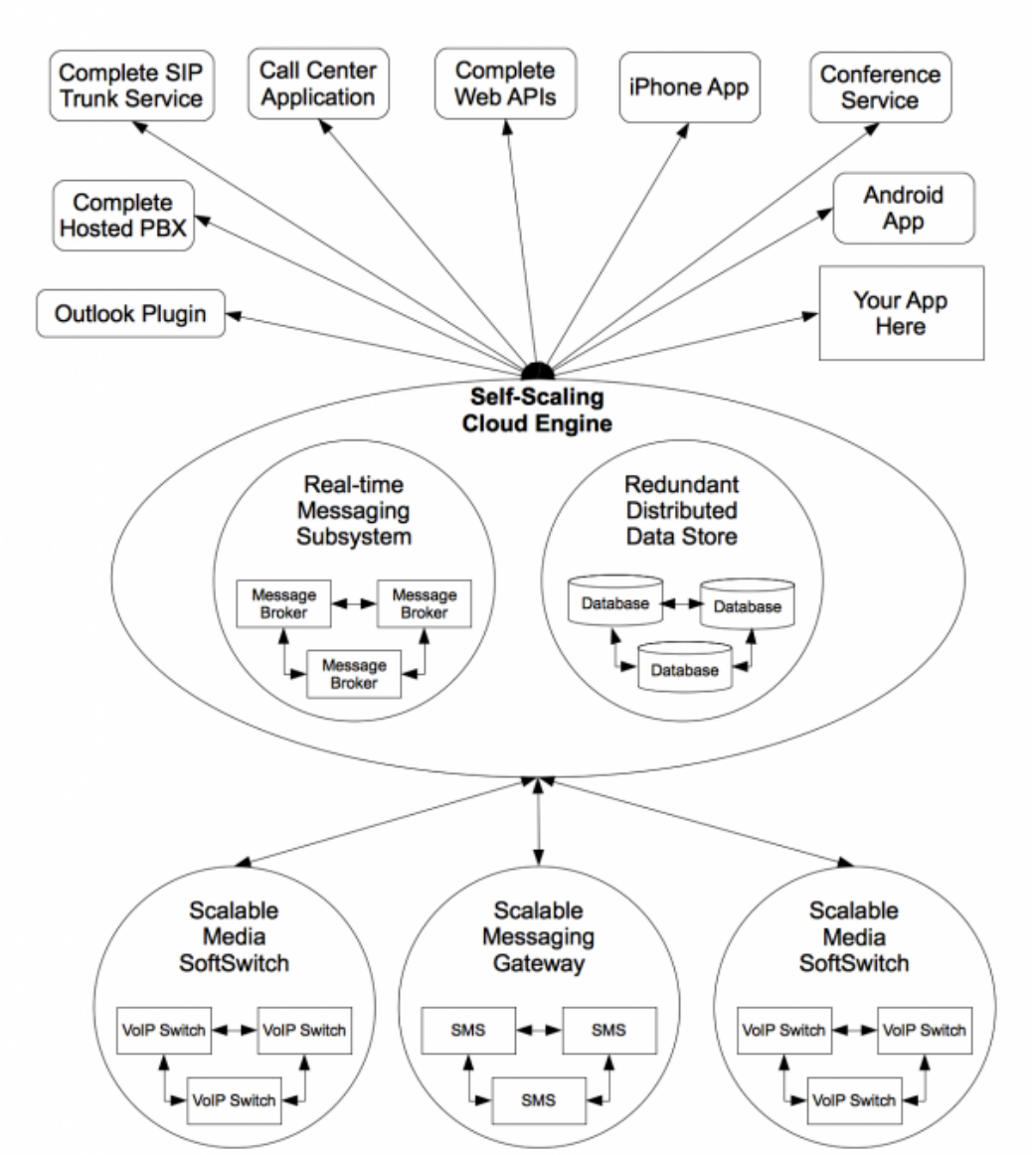
We chose the latter. Since doing so, Erlang's popularity has grown, and we're convinced it was the right choice to get us moving in the right direction.

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


whistle_amqp_sup
Whistle Applications
whistle_couch_sup
amqp_mgr
couch_mgr
AMQP
Couch
eCall Manager
Key
Supervisor
Dynamic Process
Medium Priority
Important
Process
Dynamic Process
Low Priority


ecallmgr_call_events
UUID-specific
ecallmgr_call_control
UUID-specific
For each leg of a call,
an ecallmgr_call_events
process will exist.
The events process may
or may not be linked to a
control process for the
same UUID.
ecallmgr_call_event_sup
ecallmgr_call_control_sup
ecallmgr_call_sup
start_event_process/4
start_control_process/3
ecallmgr_sup
resource_mgr
ecallmgr_media_registry
ecallmgr_fs_handler
add_fs_node/1
ecallmgr_fs_route
ecallmgr_fs_auth
ecallmgr_fs_node
lookup_route
lookup_user
Key
Supervisor
Dynamic Process
Medium Priority
Important
Process
Dynamic Process
Low Priority


Complete SIP Trunk Service
Call Center Application
Complete Web APIs
iPhone App
Conference Service
Complete Hosted PBX
Android App
Outlook Plugin
Your App Here
Self-Scaling Cloud Engine
Real-time Messaging Subsystem
Message Broker
Message Broker
Message Broker
Redundant Distributed Data Store
Database
Database
Database
Scalable Media SoftSwitch
VoIP Switch
VoIP Switch
VoIP Switch
Scalable Messaging Gateway
SMS
SMS
SMS
Scalable Media SoftSwitch
VoIP Switch
VoIP Switch
VoIP Switch

Chapter 7

Personal Insights

7.1 Learnings and Personal Achievements

learned erlang developed module on whistle managed tickets and priorities working with other people, =/= cow boy coder testing code open source technologies important in the silicon valley learning tools used by the community

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.

[8] Wikipedia. Erlang (programming language).