hypernumbers

A White Paper: Using Erlang In A Web Start-Up

Preamble

This White Paper was originally published anonymously on Philip Wadler's blog[1] in the summer of 2005 where it was called Erlang Service Architectures.

The work in it informed our previous start-up (called vixo.com) which was a social network using SMS instead of web technology.

We built the entire proposition using the precepts discussed herein. Were I to rewrite it I would remove the references to Eddie, but apart from that it stands up very well to the actual experience of building a commodity-priced Google-scale application in Erlang.

The business proposition behind that was that at the time Yahoo could monetize pages at 0.03¢ per page whereas SMS services were monetising at 2¢ - 20¢ per message. So if you can find a model that works well on the web and make it work in SMS you can coin it.

Social Networks on SMS however proved to be a dead end – at least for us – we were unable to get any uptake. If you have an immediate need for such a system tested and written in Erlang we have the source code and are open to almost any reasonable offer[2].

The irony of course is that the name vixo.com is becoming increasingly valuable. It appears to be the only tangible output of 18 months work. But the real value came from the experience of building software in Erlang in anger.

About The Author

Gordon Guthrie is the CEO/CTO of hypernumbers.com

[1] http://wadler.blogspot.com/2005/06/service-architectures.html
[2] We might even open source it – better to be used than dumped. It has all the weaknesses of a first major system written in a new language – particularly one that has not had the disciplines of real production rewrites.

1 Introduction

This document outlines the service issues in the implementation of conventional n-tier physical architectures.

2 Purpose

The purpose of this document is to outline the implementation (and profitability) consequences of software design domains for the construction of enterprise software applications – with particular regard to the selection of Erlang/OTP for such a task.

3 Scope

The scope of this document is a 3-tier logical application architecture – subject to the following assumption:

- the client interface is a browser[3] – that is to say it performs no business logic or validation[4]

There is a clear difference between two separate domains:

- -ality
- -ilities

-ality (ie functionality) pertains to the ability of a piece of software to perform certain tasks and is a user-facing process. "Does your software do this, enable that, allow the user to do the other…?"

By contrast the –ilities pertain to the management of the whole system, they are:

- scalability
- reliability
- manageability
- changeability
- securability
- performability

The scope of this paper is the –ilities and not the –ality.

[3] canonically, but not necessarily, a web browser running on some class of 'personal computer'
[4] web applications typically offer javascript validation of input data for the users convenience, but they revalidate all input at the server side – the client is not trusted to provide not invalid input

4 Table Of Contents

5 Table Of Figures

6 Quality Statement

This document has not been subject to any review.

7 Relationship To Other Documents

This document has no relationship to any other documents.

8 Structure Of This Document

This document contains the following sections:

Section	Description
Introduction	an introduction to the problem domain
Definitions	a definition of terms used in this document
Representative Distributed Enterprise Architectures	a description of state of the art enterprise architectures
Implied Foss[5] Architectures	a description of enterprise Foss architectures
Erlang/OTP Architecture	a description of Erlang/OTP architecture
Business Drivers	an outline of the business drivers and the business architecture that underpin the choice of Erlang/OTP as a strategic development language

9 Introduction

This document will look at the environment in which enterprise architectures are designed for web-based applications.

By definition 'enterprise architectures' is a protean subject, therefore this document starts by defining some representative architectures which can then be discussed in detail. The components of these architectures are drawn from experience.

Having defined some architectures this document will then compare and contrast a representative enterprise architecture with its 'mainstream' open source cousin and then an Erlang/OTP architecture.

It does this by first describing them in detail and then move onto to the comparative section under the general heading of *Business Drivers*. The key here is to locate the comparative discussion in a formal business environment not the endless aesthetic flamewars of the language communities.

[5] Foss – free and open source software

10 Definitions

10.a Introduction

This section will make explicit the terms used throughout the rest of this document. The terms that are to be defined are:

- enterprise
- logical application architecture
- physical application architecture
- test environments

10.b Definition Of Enterprise

Enterprise is used as a portmanteau term for:

- big systems
  - with a lot of users
  - with a lot of data
- software environments which require stringent –ilities:
  - reliability
  - changeability
  - manageability
  - scalability
  - availability
  - performance

10.c Definition Of Logical 3-Tier Architecture

A 3-tier logical architecture is defined as shown in the diagram below:

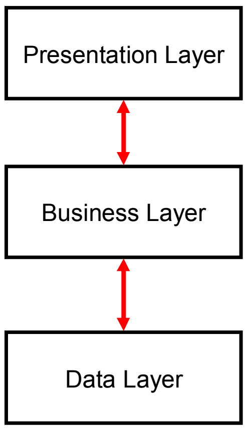

Diagram 1 – 3-Tier Logical Architecture

There are 3 defined layers:

Layer	Definition
Presentation	this layer performs no business logic, but merely marshals the information in the application and prepares it for presentation to the human user in an appropriate format
Business (or Application)	this layer is the meat of the application. It performs all the calculation and contains all the algorithms and process knowledge required for the application to run
Data	this layer performs two functions. It firstly persists and restores the data that the business layer needs to and from non-volatile storage. Secondly it tests new data, or changes to the existing data for internal consistency against a defined internal data model checking that records representing different entities maintain predefined entity relationships with each other and ensuring that the contents of fields within particular records meet the semantic requirements of the published data schema

10.d Exposition Of Physical n-Tier Architectures

10.d.i Introduction

Logical 3-tier architectures can be deployed onto a variety of n-tier physical architectures. There are 3 'classic' configurations and an infinite variety of non-classic ones. The three classic ones will be discussed here – they are:

- monolithic
- standard Lamp[6]
- enterprise

10.d.ii Monolithic

A monolithic architecture has a dumb client talking to a single physical system. The archetype is a mainframe session.

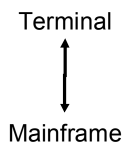

Diagram 2 – Mainframe Monolithic Architecture

[6] Lamp is the conventional name for the most common Foss architecture. It derives from Linux, Apache Mysql and Perl. In now covers a number of variants – Postgres for MySQL, Python, PHP and (less felicitously) Ruby for Perl.

The other common monolithic architecture is a standard web-based system accessing an application running on a 'web server' as shown below:

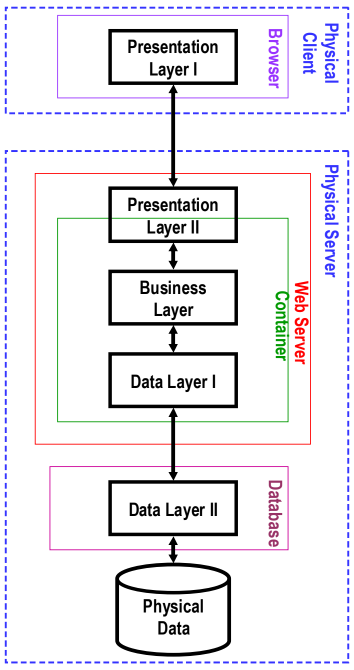

Diagram 3 –Entry Level Foss

It is worth stepping through this architecture from top to bottom. The first thing to note is that there are two distinct components of the presentation layer and 2 distinct components of the data layer.

The client user a web browser which interprets a combination of Javascript and HTML to provide a client interface – this implements part of the presentation layer. In practice it implements a sub-set of the business layer in the form of the data validation but this must be replicated with the business layer so that component is being ignored in this model.

The HTML is served to the client from a single physical machine. That runs a web server (usually Apache) which handles the HTTP protocol. Within Apache there is some sort of mechanism that executes the business logic within a container. This container can be brutally minimal as in CGI which runs the business logical within a shell, or a sophisticated container like mod_perl, mod_python, mod_php, mod_ruby or OC4J which run embedded language interpreters for (*quelle surprise*!) perl, python, php, ruby or java respectively.

Some aspects of the data layer are embedded in client libraries for these various languages, and the rest of the data layer is deployed within a 'database' – a stand alone piece of software.

The logical presentation layer is split over the client and the physical web server. The logical data layer is partly implemented in the main application container and partly in the database software – by means of constraints, triggers, semantic limitations on the permissible values of fields and other validation mechanisms.

This architecture is normally out of the box and entry level. A user emerges a standard server from a distribution. The standard server includes the web server, the embedded container, the database libraries for that container and the database of choice. The user simply writes some code and goes.

10.d.iii Standard Lamp

The standard Foss option is normally referred to as Lamp and is a physical variant of the above over two physical tiers (ie excluding the client tier) as shown below:

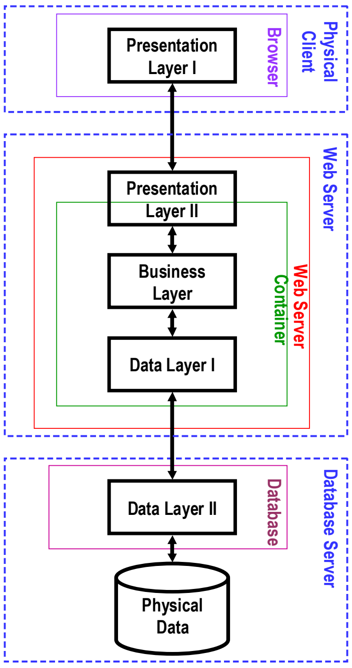

Diagram 4 – Standard Lamp

The advantage of this architecture is that it can be deployed in a (partly) redundant manner with multiple 'web' servers and a 'single' database server as shown below:

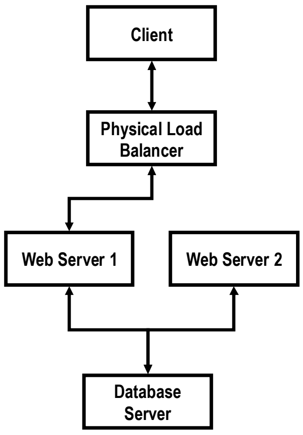

Diagram 5 – 'Resilient' Deployment

The physical load balancer allocates a connection from a user to a particular web server. Ideally it will poll for the web servers and migrate users from one server to another if one of the web servers goes down. The management of session is handled in one of two ways:

- by the load balancer
- by the application

Due to the stateless nature of the web session is usually maintained in a cookie with a validity for a period of time. That identity of that cookie acts as a pointer to a row of a server-side session table.

That cookie must be validated by the server. Normally it consists of a session ID in the form of a cryptographically-sparse hash[7].

State is persisted at the back end database. Normally, as an optimisation, session is cached at the web server and trickled back to the database server. Depending on the actual application (or more precisely its standard session library) the application is either able to manage cached state or not.

If the application cannot manage cached session then the load balancer must allocate all requests by an individual to a particular web server – so called 'sticky session' load balancing. A new request can be allocated by some other mechanism[8]. If the application can manage cached session the load balancer can push subsequent requests from a single client via any web server.

The normal use of this architecture is not high-availability. Failure of a web server is not the end of the world, but failure of the database server is. Typically this architecture gives extra resilience by increasing the number of concurrent HTTP requests that the system can handle. Apache typically starts wigging out with low hundreds of open connections and multiple web servers (albeit with low utilisations) is the canonical way around this limitation.

10.d.ivEnterprise

A typical enterprise architecture is shown below. Unlike Foss architectures it is based around Java with the core container being EJBs running under a J2EE server.

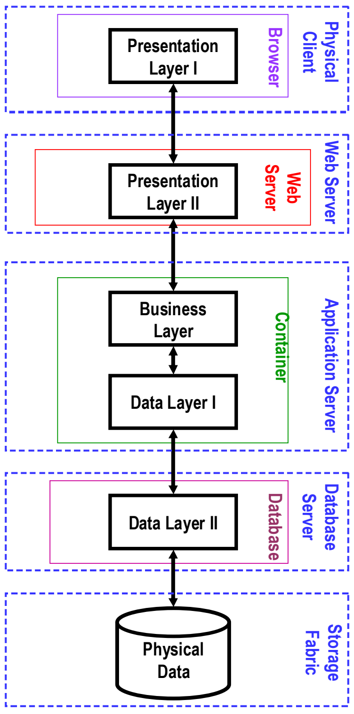

Diagram 6 – Enterprise n-Tier

Typically in an enterprise regime the data may be stored on some class of storage fabric, the deceptively similar San (Storage Area Network) or Nas (Network Attached Storage) depending on the nature of the data read-write pattern.

[7] the session ID is a number generated in some manner by an algorithm that has a low probability of collision – ie I cannot use my knowledge of the algorithm to guess a session ID that corresponds to someone else's session and is currently valid
[8] round robin, least load, etc, etc

This architecture is fit for high-availability in one of two configurations. The first is as shown below:

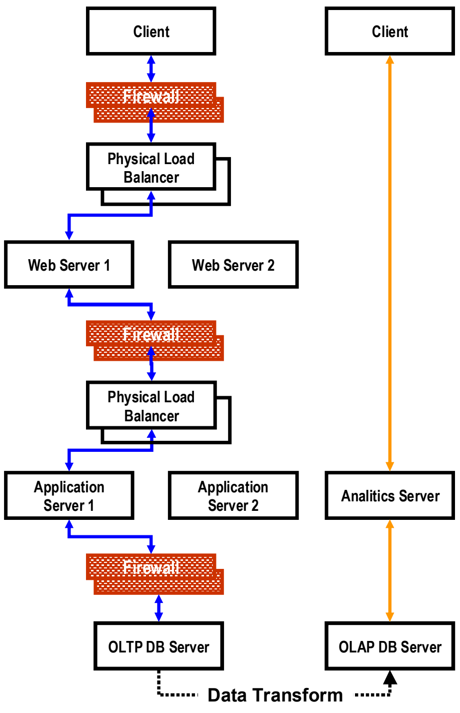

Diagram 7 – High-Availability Enterprise Architecture I

In this option high availability is provided by the following mechanisms:

- paired firewalls with failover
- paired load balancers with failover
- multiple web servers failed over by the load balancers
- multiple application servers failed over by the load balancers

There are a number of points to note about this architecture:

- it presumes that the client access if over the internet – hence the (seemingly superfluous) number of firewalls. The principle of bastioned security pertains – any single compromise by an external party should not enable them to get unrestricted access to the data. Given that most application servers[9] attach directly to the database with a pooled, high permissions account the data is effectively lost if the app server is compromised. Due or triple external physical firewalling (backed up by server-side firewalling) is designed to force a minimum of 2 compromises before data is at risk
- diagram 6 presumes a separate physical storage fabric layer which is not shown here – it is wrapped up in the 'database' server
- there are two copies of the database shown an Online Transactional Processing (OLTP) one which is normally expected to be 3rd Normal Form and is the master. In enterprise architectures it usually has an Online Analytical Processing (OLAP) clone which is heavily denormalised in a star format. The master is typically dumped and transformed on a daily basis for the management information systems to run

[9] some Oracle apps operate under a permissions-based regime where the application session acquires permissions to read and write only certain views and rows of data, but by and large application servers have full CRUD rights on the contents of tables - although usually not CRUD on tables or databases.

- there would normally be a full security architecture around the analytics stack but this is not shown in this diagram for ease of exposition

In the architecture shown in Diagram 7 high-availability is provided differently for the various physical layers:

- the web and application servers scale out – add extra servers
- the database server scales up – buy more expensive high-availability hardware

Scaling up is expensive. Top end enterprise servers (eg the Sun E20K[10] and E25K[11]) don't come cheap! Single points of failure are avoided by having multiple system boards with multiple CPUs each in a single logical operating system domain. The chassis and the operating system mean such a logical domain can survive the loss of a system board. Each logical domain is bound to multiple data interconnects providing redundant routing to the storage and the application servers. The chassis is provided with multiple power interconnects, and so on and so forth)

The other (not yet very common) architecture is shown below:

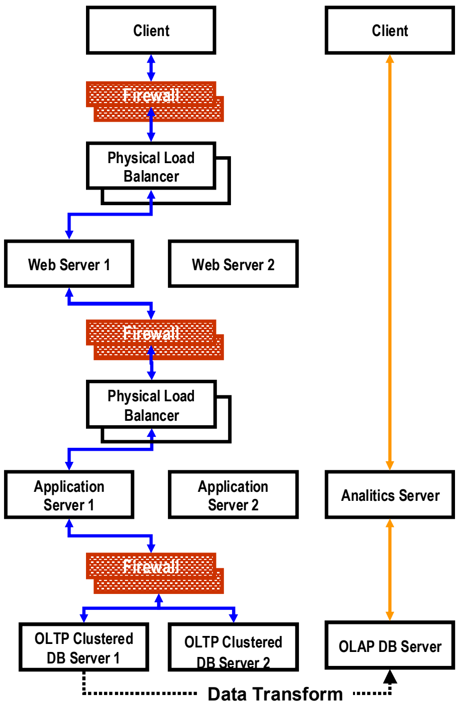

Diagram 8 – High-Availability Enterprise Architecture II

In this configuration the database server scales out (instead of up) using database clustering and table partitioning technologies (see this[12] Oracle whitepaper for instance) – to this end, in the diagram the flow of control from the client ends on not one, but two, database servers.

This section has focussed on hardware clustering through the use of enterprise class load balancers[13], but there are other clustering options available.

[10] http://www.sun.com/servers/highend/sunfire_e20k/index.xml
[11] http://www.sun.com/servers/highend/sunfire_e25k/index.xml
[12] http://www.oracle.com/technology/products/database/clustering/pdf/TWP_RAC10gR2.pdf
[13] http://www.cisco.com/en/US/products/hw/contnetw/ps792/products_white_paper09186a0080136856.shtml

One popular type of software 'clustering' is that provided by Oracle apps. They offer a load-balanced implementation where each user connects initially to a web page on a dedicated server, and are redirection to an application server for the duration of their session. The redirection is controlled by a small set of heartbeat software that 'knows' which application servers are available and how loaded they are.

Another form of software clustering is offered by modern versions of Microsoft server operating systems and the open source UltraMonkey[14]. This is based upon passive response to tcp requests. Multiple machines all listen on the same IP address. These communicate along a private backplane (a Lan or over serial cables if there are only a few of them) and negotiate between themselves which one will respond to a particular request. This may be by IP partitioning (you respond to all IP addresses 'below' 126.255.255.0 and I'll do all those above – neither of use doing 127.0.0.1 or any RFC1918[15] Ripe range).

If any machine fails to respond to heartbeats over the private Lan, the remaining ones hold an election, delegate a leader and redistribute the traffic.

10.e Test Environments

10.e.i Introduction

This sub-section will look at the logical and physical test environments for application development.

10.e.ii Logical Test Environments

The logical test environments are the path that a particular version of software goes through from the developer's fingertips to production use. It represents the logical things that are tested and the dependencies between them. The logical test environment is largely system independent and looks like:

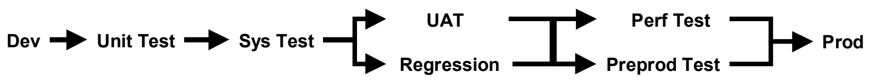

Diagram 9 – Logical Test Environments

The various environments can be defined as:

Environment	Definition
Dev (or Development)	where the developers live – where the software doesn't work
Unit Test	where the software is tested in isolation – does component X behave as components X is supposed to work
Sys Test (or System Test or Integration Testing)	where the software is tested with other components to ensure that the whole system works
UAT (or User Acceptance Testing)	where the functionality under change is tested to see if it is as expected
Regression Testing	where the functionality not under change is tested to see if the change has had unexpected consequences

[14] http://www.ultramonkey.org/
[15] http://www.faqs.org/rfcs/rfc1918.html

Environment	Definition
Performance Testing	where the system is tested to see if it can handle the load
Pre-production Testing	where the system is tested with production clone data, under production type loads and with full production configuration to see if it works as a whole
Production	where the user uses the software

10.e.iii Physical Test Environments

There are different physicalisations of the logical test environments. The/a ‘canonical’ corporate one has the following characteristics:

- automated build system performing Unit, Sys and some aspects of Regression Testing
- combined environment with UAT and the rest of Regression Testing
- single production clone environment with production data upon which Performance and Pre-production Testing is performed

The key element about this architecture is that the logical *and physical* configuration of the production system needs to be propagated back down the test environment path. If the system is hardware clustered in production then the Pre-production and/or Performance environment, and hopefully the System Testing environment need to be hardware clustered. For disaster recovery the production and non-production environments may need to be duplicated as well[16].

Pure open source software typically represents this logical environment by a simple tripartite source code tree:

- stable
- testing
- unstable

To the first approximation:

- stable = production
- unstable = dev
- testing = everything else

although automated build and test systems, use of test tools and eXtreme programming techniques may mean that some of Unit and Sys Test is built into the unstable branch.

[16] At a major UK bank there are a mandated 5 environments before production and the entire production data centre (including test environments) is replicated 15 miles way. To implement a new web server you put in a purchase order for 12!

11 Representative Distributed Enterprise Architecture

11.a Introduction

This section is structured to address the economic consequences of a (representative) distributed enterprise architecture.

It contains the following sub-sections:

- technical components
- skills sets
- development and debugging tools
- management tools

The representative distributed enterprise architecture is as per Diagram 7.

11.b Technical Components

The technical components of the representative distributed enterprise architecture are shown below:

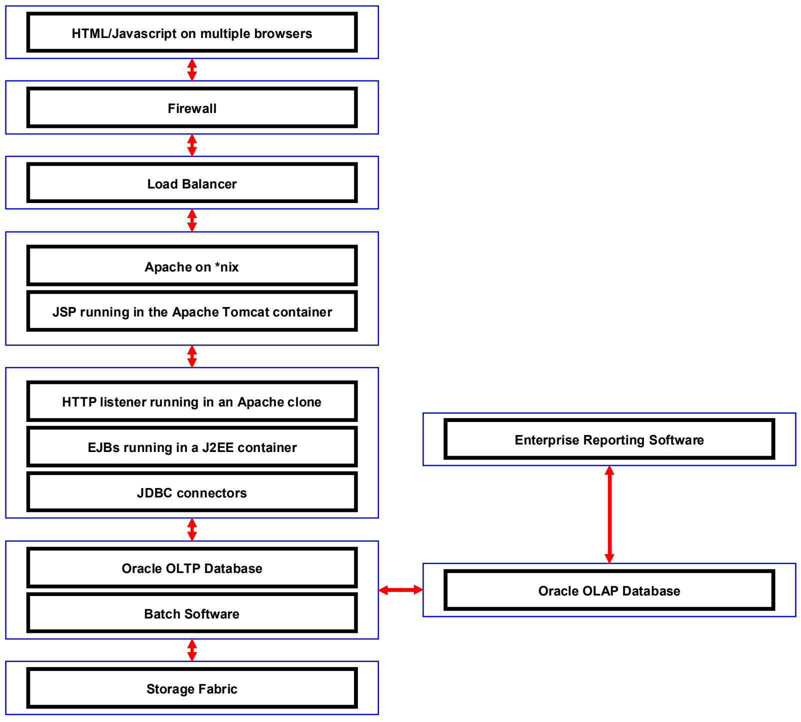
HTML/Javascript on multiple browsers
Firewall
Load Balancer
Apache on *nix
JSP running in the Apache Tomcat container
HTTP listener running in an Apache clone
EJBs running in a J2EE container
JDBC connectors
Oracle OLTP Database
Batch Software
Storage Fabric
Enterprise Reporting Software
Oracle OLAP Database

Diagram 10 – Representative Distributed Enterprise Architecture Technical Components

There are two points to note about this diagram:

- physical boxes are shown in blue – components within the same blue box are running on a single physical machine
- repeated Firewall/Load Balancer combinations between different physical layers are not shown

11.c Skills Sets

There 3 separate groups of skills sets required to build and run such an architecture comprising:

- design
- development
- support

11.c.i Design Skills

To design such a system, knowledge is needed of:

- HTML
- Javascript
- DNS[17]
- NTP[18]
- networking
- JSP[19]
- EJB/J2EE[20]
- JDBC[21]
- SQL[22]
- database design
    - OLTP[23]
    - OLAP[24]

11.c.ii Development Skills

Typically there are 4 development teams[25]:

- front-end/user experience
    - web pages
    - usability
- front-tier
    - JSP
- application developers
    - Servlets
    - EJB/J2EE
    - SQL
- database designers
    - data layout
    - SQL profiling
    - OLTP/OLAP transform
    - report writing

---

[17] Domain Name Service
[18] Network Time Protocol
[19] Java Server Pages
[20] Enterprise Java Beans and Java 2 Enterprise Edition
[21] Java Database Connectivity
[22] Structure Query Language
[23] Online Transaction Processing
[24] Online Analytical Processing
[25] historically enterprise development is silo-ed into skill based teams

11.c.iii Support Skills

The following support skills are required:

- platform support – ideally only one operating system and version within the data centre – however this is often not the case
- application infrastructure support
  - web servers
  - J2EE container
  - database software
  - DNS servers
  - NTP servers
  - reporting tools
- network support
  - firewalls
  - load balancers
  - switches (not shown on the diagrams)
- hardware support
  - low-end *nix servers
  - high-end *nix servers
  - storage fabric and switching

11.d Development And Debugging Tools

Typically the following set of tools will be required:

- Dreamweaver for HTML/Javascript development using the native script debugger
- JBuilder or Forte for JSP
- JBuilder/NetBeans/Eclipse for JSP/servlet/EJB development
- JUnit[26] test suite for JSP/servlet/EJBs
- some class of dynamic test tool (eg Winrunner or Winloader)

The key point to note is that the testing and debugging tools are only cover part of your whole deployment. It is possible to do remote debugging with Java whereby you attach to a remote JVM with a debugger, but it gets progressively more complex to track a single end-user activity across a distributed n-tier environment. The debugger cannot follow the flow of execution from one node to another. You cannot do proper real-time n-tier debugging – anything on n-tiers requires synchronised printfs and comparison of log files (having checked the server times are in synch).

One of the other catches is that different tools have different strengths for different types of 'java' development, one team may want to use JBuilder, a second NetBeans and a third Eclipse…

11.e Management Tools

Typically such an installation would be managed with the following consoles:

- a network console which would be capturing network, switch and load balancer SNMP traps
- a hardware console from the manufacturers which would report on hardware failures for each manufacturers hardware – ideally there would be one manufacturer only
- an Oracle Database Management Suite console for database management

[26] http://www.junit.org/index.htm

- a hardware/infrastructure software console (eg Big Brother[27]) which the platform support team would write bespoke traps for

In addition there may (or may not) be a single logging host which all platforms and applications are configured to write to. The physical systems should be taking a common time feed from an NTP server and therefore the different log files should all be timestamped with an appropriately shared timestamp – giving the ability to map problems.

Some log-to-SNMP automated processing jobs may be written which scan the log files and pop appropriate alerts due to particular types of messages onto the console.

The source code for the application components should be under a common source code management system (although this is often not the case due to the fact that different toolsets for different components have different integration with source code management). In addition the source code management should integrate with an application build system that can generate application binaries against an infrastructure software and platform version. The whole (application, infrastructure software and platform) should be deployable under a common configuration/change management and deployment system.

[27] http://www.bb4.org/

12 Lamp Architecture

12.a Introduction

This section is structured to address the economic consequences of an implied Foss architecture.

It contains the following sub-sections:

- technical components
- skills sets
- development and debugging tools
- management tools

There are no effective standard reporting tools in the Foss space (or if there are, I am unaware of them) and this discussion will therefore merge OLAP and OLTP considerations into a single generic component stack.

12.b Technical Components

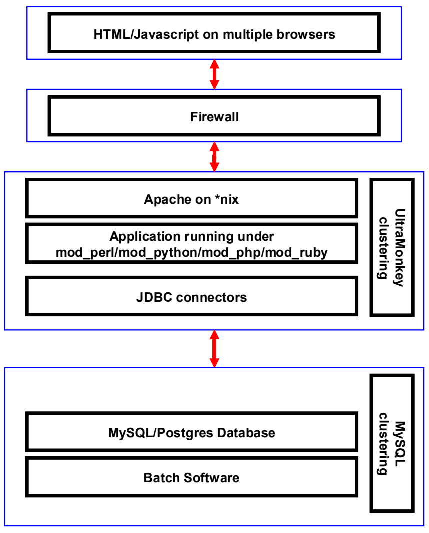

Diagram 11 – Implied Foss Architecture Technical Components

The key issues to note there are that:

- both UltraMonkey and MySQL clustering is hot-hot and automatic, all the components are running and failover happens without human intervention

12.c Skills Sets

12.c.i Design Skills

To design such a system, knowledge is needed of:

- HTML
- Javascript
- DNS
- NTP
- Networking
- Apache
- a Lamp language and mod_*lamp*
- *lamp*:DBI[28]
- SQL[29]
- database design
    - OLTP[30]
    - OLAP[31]

The skills required for the Lamp language development include a detailed understanding of the templating mechanism required to integrate functional HTML output with the business logic – a separation that is implemented across two separate physical tiers in the representative distributed enterprise architecture.

12.c.ii Development Skills

Typically there are 3 development skill sets (which may or may not be realised as different teams[32]):

- front-end/user experience
    - web pages
    - usability
- application developers
    - Apache and mod_*lamp*
    - *lamp* language and templating system
    - SQL
- database designers
    - data layout
    - SQL profiling
    - OLTP/OLAP transform
    - report writing

[28] ie the Database Interface for the appropriate Lamp language – perl:DBI (http://dbi.perl.org/), Python DB-API 2.0, PHP:DB (http://vextron.mirrors.phpclasses.org/browse/package/156.html) or Ruby:DBI (http://ruby-dbi.sourceforge.net/) – note I have only used the Perl and Ruby versions and am not familiar if the Python or PHP ones are the canonical mechanisms for database access in those languages

[29] Structure Query Language

[30] Online Transaction Processing

[31] Online Analytical Processing

[32] the nature of the open source market – typically with developers coming through from implementing systems in a monolithic architecture - means that there is much more often a wide spread of skills throughout individuals. At the low end a single developer will be responsible for all the development across the logical tiers as well as the deployment, security and support of the server.

12.c.iii Support Skills

The following support skills are required:

- platform support – ideally only one operating system and version within the data centre – however this is often not the case
- application infrastructure support
    - apache with mod_*lamp*
    - MySQL
    - UltraMonkey clustering
    - MySQL clustering
    - DNS servers
    - NTP servers
- network support
    - firewalls
    - switches (not shown on the diagrams)
- hardware support
    - low-end *nix servers

It should be noted that there are limitations on this model around the database cluster. MySQL clustering is only for availability – not distribution. The distributed data is the same for all the nodes in the cluster (except that some are in memory and some are on disk with synchronous slaved log writes – the whole read traffic being handled from in-memory). Therefore there is a scalability issue – if the total database gets very big then each machine in the database cluster needs to scale up – scaling out is only for reliability.

12.d Development And Debugging Tools

The development and debugging tools are two varied to look at in detail here, however there are a couple of salient points to be made about them:

- most of the *lamp* languages have access to quite sophisticated IDEs and debuggers, some of them even having remote debugging facilities as per the Java case in Section 11.d – however these are subject to the same limitations discussed therein
- most of the *lamp* languages have a comparable implementation of JUnit for instance Ruby Unit[33].

12.e Management Tools

The management toolkit is very similar to that in Section 11.e – there is the clear distinction between the various hardware and network consoles and the application logs with any integration being done via log-to-smnp monitoring.

[33] http://www.ruby-doc.org/stdlib/libdoc/runit/rdoc/

13 Erlang/OTP Architecture

13.a Introduction

It contains the following sub-sections:

- technical components
- skills sets
- development and debugging tools
- management tools

The Erlang/OTP enterprise architecture is shown below and can be through of as a version of Diagram 9 – High-Availability Enterprise Architecture II – with the elision of the enterprise OLTP and OLAP data and reporting architectures.

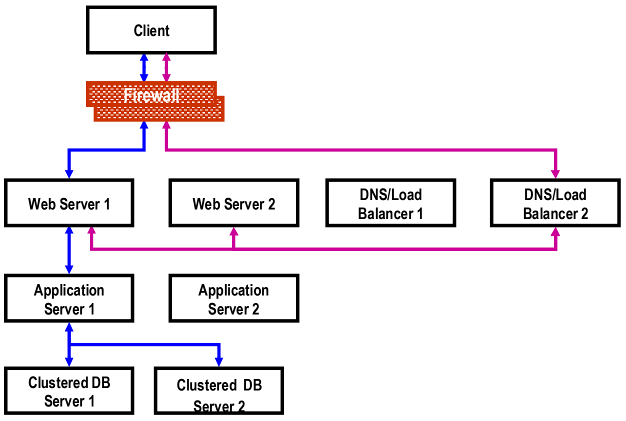

Diagram 12 – Erlang/OTP Enterprise Architecture

There are a number of points to note here:

- the Erlang/OTP cluster comes from a world in which access to the networks is physically and logically limited – the phone network. Erlang Virtual Machines on the same cluster implicitly trust each other. If a bad guy has access to an Erlang shell on one of them, he has the full rights over all the Erlang VMs in the cluster – hence the architecture diagram doesn't show any firewall between logical servers[34]
- software clustering is provided via a clustered, load-balancing DNS to the client

13.b Technical Components

Prior to discussing the technical components it is important to outline the Erlang/OTP method of constructing an application.

There are three components to an Erlang/OTP application[35]:

- applications
- supervisors
- workers

[34] this is the single biggest concern I have about the deployment of an enterprise Erlang cluster connected to the internet. The solution will probably involve fronting the whole cluster with an application reverse proxy which uses a set of rules generated from the information architecture implied in the application source code to build a Level 6 filter on all incoming requests as well as some class of Level 6 filter on outgoing traffic…

[35] for more details see http://www.erlang.se/doc/doc-5.4.3/doc/design_principles/part_frame.html

A worker is a piece of code that performs activities and is presumed to be error prone. It is in a hierarchical child-parent relationship with a supervisor (the worker is the child, the supervisor is the parent).

A supervisor is defined to be part of the error kernel[36]. The role of the supervisor is to start and monitor workers and to restart them if they fail – as well as writing away the error as a bug.

A supervisor can have another supervisor or an application as a parent and another supervisor or a worker as a child.

An application is a collection of applications and supervisors and knows how to start them, in what order they should be started and how to restart them if they fail. It knows how to start its processes and applications across a range of logical Erlang Virtual Machines and how to fail them over if those machines become unavailable.

This model is shown below:

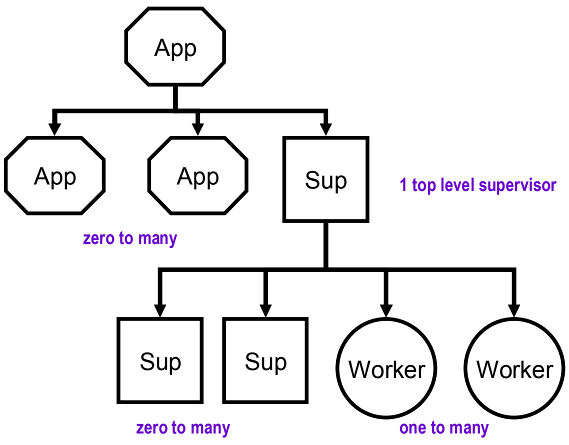

Diagram 13 – Generic Erlang/OTP Application Structure

From this perspective the technical components can be shown as:

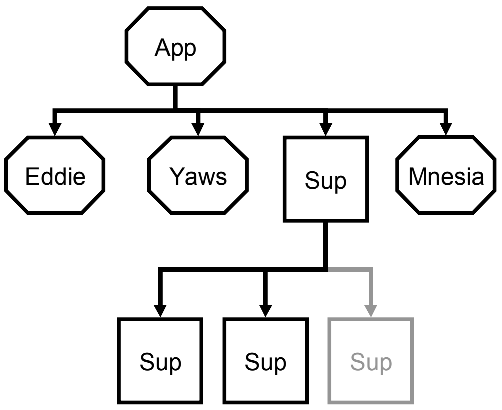

Diagram 14 – Erlang/OTP Web Application Structure

[36] http://www.erlang.se/doc/programming_rules.shtml#HDR15

The three embedded applications are:

- Eddie[37] – a load balancing DNS server
- Yaws[38] – yet another web server – with a configuration based on Apache
- Mnesia[39] – the Erlang/OTP distributed database

13.c Skills Sets

13.c.i Design Skills

- HTML
- Javascript
- DNS
- NTP
- Erlang/OTP (including eHTML)
- networking
- database design

13.c.ii Development Skills

The skill set is much condensed compared to the other defined stacks:

- front-end/user experience
  - web pages/HTML/eHTML
  - usability
- application developers and database designers
  - Erlang/OTP

13.c.iii Support Skills

There are two components to the support set:

- platform support
- application support

The key thing to notice, however, is that much of what is considered infrastructure application support (web servers, databases, DNS, etc, etc) is actually integrated into the application and it the responsibility of the development team.

13.d Development And Debugging Tools

Erlang/OTP comes with a full set of development tools (see for instance the section entitled Tool Applications here[40]):

- a real n-Tier debugger[41]
- an emacs mode and embedded Erlang node (Distel[42])
- n-Tier tracing

[37] http://eddie.sourceforge.net/news.html
[38] http://yaws.hyber.org/
[39] http://www.erlang.se/doc/doc-5.4.3/lib/mnesia-4.2/doc/html/index.html
[40] http://www.erlang.se/doc/doc-5.4.3/doc/
[41] http://www.erlang.se/doc/doc-5.4.3/lib/debugger-2.3.1/doc/html/index.html
[42] http://fresh.homeunix.net/~luke/distel/

In addition Erlang/OTP (the language) understands how to replace running code. OTP compliant modules have a native understanding of what version they are and how to upgrade themselves on the fly to the new version of code without stopping the running application at all.

13.e Management Tools

Given that one of the core principles[43] of Erlang/OTP programming is to entirely eschew defensive programming and to allow workers to fail, it is not surprising that Erlang/OTP has excellent capabilities in error reporting.

There is a native SNMP library which can be used to represent errors as SNMP trappable events. Strategically the entire application should report as a common set of errors and be integrated into an SNMP console along with the platform and networking hardware and enable a coherent, entire application management framework to be created that enables the system to be holistically managed.

[43] http://www.erlang.se/doc/programming_rules.shtml

14 Business Drivers

14.a Introduction

A profitable enterprise application has a lifetime measured in years. The overwhelming component of its cost to an enterprise comes in the support and maintenance phase of its life and not the development phase.

This section will look at the lifetime cost model of an Erlang/OTP application versus a conventional enterprise or Lamp one.

It will consist of the following sub-sections:

Sub-Section	Description
philosophy	a summary of the philosophy behind: • a balanced cost/revenue curve • a low service-cost global enterprise system
scalability	a description of the cost issues in scaling
reliability	the costs and trade-offs in reliability
manageability	the management of a systems and the costs embedded in simply feeding and watering an enterprise application
changeability	the management and testing of change to an in-service application and its cost impact
securability	how the system is secured against intrusion and the costs thereof
performability	how they system is made to perform to an appropriate standard

14.b Philosophy

14.b.i Introduction

There are two philosophical components which have an overwhelming bearing on the selection of Erlang/OTP as the enterprise development platform and which are best described in isolation.

14.b.ii A Balanced Cost/Revenue Curve

The key philosophy here is to tightly couple the cost of scaling up with the revenue to be gained from scaling up. Business growth can be caricatured as a smooth curve and infrastructure costs as a stepped line. The key is to have similar granularity in change of those curves as shown schematically below:

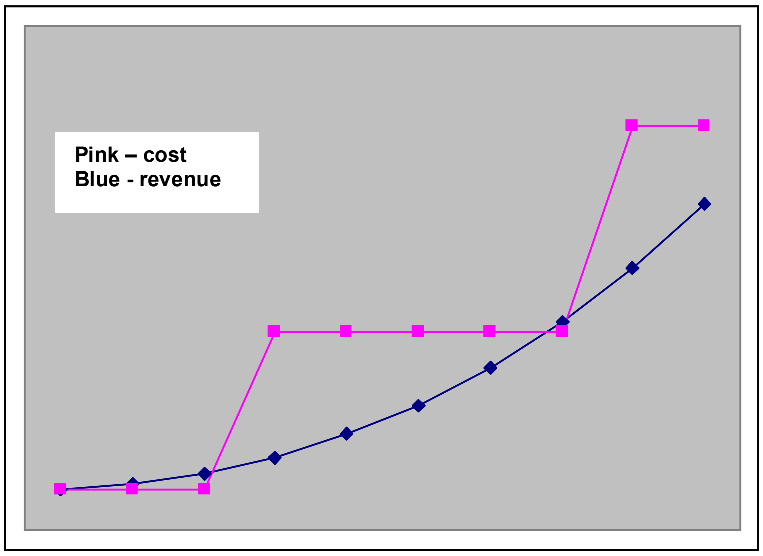

Diagram 15 – Unbalanced Cost/Revenue Curve

14.b.iii A Low Service-Cost Global Enterprise System

The philosophy behind low service-cost can be metaphorically described in terms of Ryan Air versus British Airways – the elimination of difference:

- one set of development tools
- one source code and configuration/build management tool
- one set of developers with end-to-end reliability
- real-time n-Tier debugging and tracing for holistic debugging and an end to blame-storming
- integrated hardware, network and application consoles for the holistic management of the system
- as few skills as possible
- fungible hardware in the production architecture – the same class of machines/builds performing DNS/web/application and database functions
- fungible hardware across the entire test environments from dev to prod

The intention is to radically address the cost of negotiation:

- how does my change control system work with yours?
- how does my version match to yours?
- how do we co-ordinate everything so that our testing is coherent?
- what is the problem?
- who is responsible for this problem?

There are understood and exploding costs associated with team size. Once a team gets above about 20 to 25 then communication within the team moves from an informal function to a necessarily formalised one.

One of the most interesting side effects of the open source revolution has been the radical standardisation of development tool sets. Neilson's Law[44] operates very strongly on open source development sites. Almost every open source component does the following things either the same or a very small set number of ways:

- accessing code
- version naming
- logical test environments
- bug reporting
- documentation generation
- unit and automated testing
- software build and install
- library access
- dependency management

Rigorous systematisation of processes and tools generates considerable benefits – bizarrely it is easier for me to join a open source software project consisting of people I have never met than it is to participate in one at work where I know everyone, I sit near them and we are all paid to work together!

One of the key insights about process and tool consolidation is that it doesn't matter that any individual component is the best (or even near the best) there is a considerable benefit to have one consistent way of doing things and one consistent tool set.

14.c Scalability

If the ambition of the firm is to make global enterprise systems with hundreds of millions of users then the application needs to scale linearly with users and the development language needs to support that.

A typical 100 million user application provides the same (small) amount of functionality to a humungous quantity of users. There are many ways to provide this but ultimately this requires some degree of partitioned data – there is no server in world that can provide a monolithic 100million user database.

Erlang is designed along 'shared nothing' lines with message passing and is inherently scalable (as a potential cost to performance).

There are other fundamental problems with scaling 3-tier physical infrastructure in an 'out-out-up' configuration as per Diagram 7 – High-Availability Enterprise Architecture I. The cost of enterprise 'up-scaling' hardware is prohibitive in a start-up environment. The granularity of that cost-space is in the order of hundreds of thousands (or indeed millions) of pounds – giving the sort of stepwise costs seen in Diagram 15.

There is a similar initial cost in providing enterprise scale physical clustering load balancing (as opposed to application or software clustering and load balancing.

By contrast a configuration as per Diagram 8 – High-Availability Enterprise Architecture II with scaling provided by Cots[45] hardware provides a much smoother curve.

[44] http://www.useit.com/alertbox/20000723.html
[45] Cots – an acronym for 'cheap off the shelf'

There is a second element to this which flows from the mapping of the logical to the physical test environments as per Section 10.e.iii. A huge proportion of the costs of servicing an application flow from supporting the testing environments – and the cost base of testing is defined by the software and hardware configuration of the production system. The ability to overload physical hardware with multiple logical test environments (today Unit Test, tomorrow UAT) enables considerable cost savings – combining that with the use of Cots hardware for production and testing is even more desirable.

Business Driver: mass market – a million biros at £1 each, not one pen at a £1m

14.d Reliability

14.d.i Introduction

Reliability is notoriously hard to achieve. Erlang/OTP has been proven to provide the highest reliability[46] in the world.

Among the key components to ensuring reliability are:
- zero defect programming
- n-Tier debugging and tracing
- hot-swappable code and hardware
- macro stability and grace under load

Business Driver: customer satisfaction – 'product dialtone'

14.d.ii Zero Defect Programming

The utilisation of a zero defect approach to software development by definition requires the abjuration of defensive programming. The application must be stable in the face of logical or transient error. The Erlang error kernel supervisor/worker paradigm provides the only way I know of doing this in a programming language.

Zero defect programming is only possible in a development architecture that supports separation of the robustness of a system from logical errors. By their nature exceptions generate encapsulation not resolution of bugs.

Business Driver: fixing bugs is cheaper than finding them – and bugs breed bugs

14.d.iii n-Tier Debugging And Tracing

n-Tier debugging and tracing directly impact on the amount of support that a system requires. Real n-tier debugging is key to being able to deliver zero defect programming – n-tier *printf* statements synchronised via NTP timestamps is not practical in complex distributed applications.

Business Driver: best use of expensive developers - new functionality generates income, fixing bugs merely protects it

[46] http://www.guug.de/veranstaltungen/ffg2003/papers/ffg2003-armstrong.pdf - see reference 10.

14.d.ivHot-Swappable Code And Hardware

Any reliable application must be able to be changed on the fly. Both the hardware and the software must be hot-swappable. Hardware that is component hot-swappable (ie highly resilient hardware) is expensive. The Erlang/OTP approach enables physical hardware to be treated atomically for the purposes of hot swap (ie bring a server on- and off-line as required) and obviates the need for high availability hardware. The inbuilt ability of an Erlang/OTP module to swap itself out on the fly enables continuously operating systems to be built.

Business Driver: reliability on Cots hardware

14.d.v Macro Stability And Grace Under Load

The concurrent nature of Erlang is the key to the macro stability of the system, and in particular grace under load. Erlang implements very cheap 'processes[47]' and happily supports 10s of thousands of them in a single threaded VM. All these processes communicate by sending each other messages with 'nothing shared'. This underlies the entire structure of the language and its ability to operate in a distributed and failover fashion. This is discussed extensively here[48] (and I cannot begin to improve on it). However it is worth considering the question of concurrency in the quoted enterprise and Lamp architectures of this document.

Greenspun's Tenth Rule of Programming states that:

> " any sufficiently complicated C or Fortran program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp."[49]

Following the lead[50] of Sven-Olof Nyström it is interesting to look at the implementation of concurrency in the reference architectures discussed above. Essentially Erlang is an elegant solution to the problem of concurrency and the cost of native threads and processes. The two reference architectures both contain multiple instances of elaborate supporting architecture designed to manage the cost of native concurrency as enumerated in the table over.

[47] Erlang processes as opposed to operating system processes
[48] http://www.guug.de/veranstaltungen/ffg2003/papers/ffg2003-armstrong.pdf
[49] http://www.paulgraham.com/quotes.html
[50] http://prog.vub.ac.be/~wdmeuter/PostJava04/papers/Nystrim.pdf

Logical Tier	Enterprise	Lamp	Notes
Client	Frames, Flash or Ajax[51]	Frames, Flash or Ajax	All these technologies break the monolithic model of web interactions by breaking down the user interface into concurrent sections and servicing these differently. All three have a distinguished history of use despite the fundamental problems they cause by breaking the uniqueness of URLs see for instance http://en.wikipedia.org/wiki/AJAX and http://www.useit.com/alertbox/20001029.html
Presentation	Apache and Tomcat	Apache and mod_*lamp*	In the beginning was Apache and it handled requests in the CGI-bin by forking a new operating system process for each request – and that was slow. Then came the module tree whereby a series of persistent language interpreters were pre-spawned at start up. These interpreters were allocated an operating system process each. The role of Apache was to manage this pool of pre-created processes by activating them and handing off HTTP requests, thereby obviating the process start up costs. Apache 2 is the solution to the problem of different threading models in underlying architecture. It allows for the pre-spawning of multiple operating systems processes with (potentially) multiple operating system threads and the whole of these double pools of resources are managed by a combination of Apache and the Apache models
Application	EJB's under J2EE	lamp:DBI	J2EE provides an environment to run EJB's in. One of the key components of the specification is to enable EJB's to natively be deployed in a pool whereby a number of uninstantiated EJB's of a particular type are created and then the container will handle the assignment of said EJB's to function calls. These EJB's are provided with automatic serialisation to ease this process. By default the data interconnect is implemented as a connection pool to ease the cost of setting up and tearing down database interconnections (of which part is the process/thread startup cost at both ends and part is authentication/permissions). For the lamp stack lamp:DBI performs the same role – providing a self-managing connection pool for the database access.
Database	Oracle	MySQL/Postgres	Databases create and manage their own thread and connection pools.

[51] http://www.adaptivepath.com/publications/essays/archives/000385.php

It is also interesting to note that the two great black arts of enterprise development are intimately linked in the provision of concurrency:

- performance tuning
- threading

In an enterprise/lamp architecture performance is 'enabled' by attempting to align the various thread and resourcing bottlenecks through the piece until the throughput is 'satisfactory'. Prudence dictates that the tuning achieved is fragile and that generous 'headroom' is left. That headroom translates into extra servers and bears directly down on the cost/income ratio. In addition there is a presumption that servers 'must' fail under load and consequently that the entire class of transient, high-load errors are simply not fixable – there is an immutable floor above zero defects.

The Erlang approach of forcing serialisation at first contact (the VM is single threaded) and then on communication in a fundamentally non-blocking asynchronous manner simply abolishes thread hell… By using stable time slicing techniques it allows systems to be specified that can support horrendous loading edge cases[52] by degrading gracefully and with minimal impact on the customer.

Business Driver: abolish expensive 'black art' activities – increase utilisation and reduce *recurring* contingency costs which come straight off the bottom line.

14.e Manageability

Management costs increase with specialisation – the marginal cost of management of each:

- platform
- test environment
- toolset
- skillset
- development and support team

Particularly in a start up phase a development team will come up against the atomicity of staff. I can be 1/3rd web designer, 1/3rd java code and 1/3rd DNS support, but I'll never get to 1/10th…

The cost of managing an infrastructure, detecting and fixing problems, measuring and monitoring systems and performance also increases with the number of consoles.

Simply identifying and categorising a problem across a multi-console system is enormously expensive.

Soup to nuts single console management is a key to zero defect programming and fast problem resolution.

Business Driver: less staff, more flexibility, lower management/developer ratio, quicker and cheaper problem resolution, better information

[52] particularly if they are of high intensity but short duration in which case they simply wont appear to the customer

14.f Changeability

By definition the business environment is unpredictable. The key aspect to change is the ability to alter the functionality delivered to the customer. The first part of that is dependant on the –ality, how easy it to change things in the development language, and is consequently out of scope.

But the larger costs of change are down to the delivery of functional change to the production environment and non-functional changes.

In order to maintain a clean and tested production environment it is necessary to take code changes through strict and sequential testing regimes which need to be automated. The cost of change is, in a large part, determined by the costs of testing – and these are heavily influenced by the costs of configuration and build management and the fixed costs of the physical testing environments. The entire mantra of less diversity in tools and techniques speaks directly to this cost base.

In addition, non-functional changeability is largely determined by the ability to scale, rescale and reorganise resources to fit a changing world. The construction of a global enterprise system out of fungible hardware components enables this degree of flexibility.

Business Driver: shorter change to production elapsed time, higher quality testing leading to less bugs-on-bugs, cheaper reorganisation of physical production and non-production hardware.

14.g Securability

Security is the Achilles heel of Erlang. Due to the trusted nature of telephony networks (at least compared to the internet) Erlang has no security. All nodes in an Erlang cluster are implicitly trusted by all other nodes and once a bad egg gets access to one, they get full access to all.

Bastion security, sacrificing front-tier machines in the name of data security, (see for instance Diagram 7 – High-Availability Enterprise Architecture I and accompanying discussion) is simply impossible. The entire Erlang cluster needs to be wrapped in another security layer to provide indirection.

Business Driver: an insecure application is a direct cost to the business.

14.h Performability

Systems need to perform, both absolutely and under load and other abnormal conditions and Erlang has an excellent track record in this respect. The old ‘interpreted versus compiled’ debates and the need to be ‘close to the metal’ have been revealed as merely ‘premature optimisation’ by the great advances in processing power. Large clusters of commodity hardware provide more than adequate performance at more than acceptable prices.

Business Driver: there are two performance drivers – unsatisfactory customer experience and low customer/server ratios – the infrastructure must be performant to the income per transaction and/or customer.