Product report: Trapexit 2.0

Yury Dorofeev, Jacob Ericsson, Hariprasad Hari, Jonas Rosling,
Niclas Stensbäck, Samuel Strand, Wilson Tuladhar, Yeli Zhu

February 28, 2011

Contents

List of Figures

Abstract

In order to invigorate the trapexit.org website, a website entirely written in Erlang 2.1 is proposed and implemented. A complex and multilayered architecture is constructed wherein loosely coupled components making up a traditional web-site (database, web server etc.) use APIs to communicate over a communication framework, implementing a bus architecture.

1 Introduction

Like most programming languages, Erlang 2.1 has many community websites, one of them being trapexit.org. It is currently managed by Erlang Solutions [15], and this projects deals with laying the groundwork for an overhaul of that website.

Making a website entirely using Erlang can be quite easy or annoying, depending on how you approach doing it. Web frameworks such as Erlang Web 2.5.1, Nitrogen 2.5.2, Zotonic 2.5.4 and others make it easy to host your own webpage based on Erlang if you wish it. What we do, though, is something quite different. In the beginning of the project an outline was made clear: our project uses layers of abstraction to make sure that all the components are exchangeable (to differing degrees). This was an approach which grew out of a process of lengthy research and discussion about our architecture, which will be described in the following chapters.

Our main goal, throughout the project, was to make a system backbone that is robust, modular and (where possible) fast. This is in accordance with the three golden rules of Erlang

- First, make it work
- Then, make it beautiful (in our case, modular and stable)
- Then, if someone is pointing a gun at you, you make it fast.

We did not get to the third bit, due mostly to time constraints, although the stress testing we have conducted have garnered some interesting results, which can be found in chapter 5, Testing. Our main result is having constructed a website that works using a bus architecture and through many layers of abstraction still manages to handle large amounts of traffic efficiently.

2 Tools

This section is about the tools and programs we have chosen to use in our system.

2.1 Erlang/OTP

One of the major highlights of the project is using Erlang [3] in all components and to connect them. Though the project team had various backgrounds and few of us had any real experience using Erlang, after going through two courses ("Erlang by example" and the aptly named course "Open Telecom Platform") we all felt rather good about using it. Obviously we haven't coded everything as an experienced team of Erlang programmers might have, but making clear and understandable code was still relatively easy.

Erlang is also a good fit due to the concurrent nature that a website backend holds. The efficiency of Erlang processes and the lightweight nature of its message passing made it easy for us to spawn processes to handle requests, to break up our dataflow into small and oversee able pieces or to just build quick proof-of-concepts using OTP.

Choosing Erlang to make a website solution in all parts may also be considered an interesting proof-of-concept, although it certainly can be done in other ways as well. What our solution explores is the highly distributable nature of an application that Erlang wasn't made for and that, perhaps, wouldn't be as natural to other web applications.

2.2 Choice of database

In the choice of database, we had three contenders to choose between: Riak 2.2.1, CouchDB 2.2.2 and Hibari 2.2.3.

2.2.1 Riak

Riak [14] is an open-source, key-value store database written in Erlang 2.1. Its main selling point is distributability, which made us wary. We knew that the main issue would not be distributability in our project.

Apart from that, Riak turned out to be cumbersome and awkward to use. When doing basic operations like reading or writing, Riak returned large, nearly unreadable blobs of metadata.

While there was a tutorial for setting up a basic cluster of nodes there was no obvious way to run Riak on a single machine and the documentation other than the initial setup was somewhere between lacking and nonexistent.

2.2.2 CouchDB

CouchDB [1] is a document store database, built by the software foundation Apache and is written entirely in Erlang 2.1. Document store entails that data is stored in "documents" without any overlying structure other than the name of the document and the data field that you input into the document.

While installing CouchDB required a hack or two to make it work in Ubuntu [16], the documentation on how to do this was fairly straightforward and available in the official CouchDB wiki. The wiki contained detailed information about how to run and configure the system.

When CouchDB was running, it was easy to use and had an accessible web interface useful for debugging and development, although it was prone to crashing occasionally.

2.2.3 Hibari

Hibari [17] proved difficult even to get running in the first place and the database seriously lacks online documentation.

After spending a day trying to make it work for us, we could barely install it and we could not make it run.

2.2.4 Decision

Hibari 2.2.3 was quickly dismissed as a candidate for us, so the real decision was between CouchDB 2.2.2 and Riak 2.2.1.

Firstly, Riak's main selling point and specialization was something that we knew that we would hardly, if at all, use in our project, distributability. CouchDB's main selling point, ease of use and a REST-based API was definitely something that we could make use of.

Secondly, Riak was cumbersome to use and we felt that it would require a lot more work to get Riak to work for us than it would take to make CouchDB to work for us.

Finally, Riak had a beginner's guide. CouchDB had a well organized wiki and a free book of 200+ pages. Again, the time constraint of one semester played a part. We could reverse engineer Riak and figure out how it worked. Or we could just read the manual for CouchDB and make it work much faster. All of these reasons taken together pointed clearly in the direction of CouchDB, and that was the database we chose for this project.

2.3 Communication Framework - RabbitMQ

Since our task was to make a website which uses a message bus for the message passing between the different components, we started to look for some message bus applications and found two of them: RabbitMQ [4] and ZeroMQ [18]. Since we had to use Erlang 2.1 built components and ZeroMQ is written in C++ [19], we chose RabbitMQ.

RabbitMQ is a message broker software which uses the AMQP(Advanced Message Queuing Protocol) [11] standard. The major features of AMQP standard includes message orientation, queuing, routing, reliability and security. RabbitMQ implements AMQP to provide a point of rendezvous between our backend systems and the frontend systems. Messages are published to the services and the services have options to subscribe or get the messages on requests. An example of how AMQP looks like can be seen in figure 1. The AMQP model has various entities such as:

- Message Broker: a server to which AMQP clients connect using the AMQ protocol
- User: all the application users that want to send message through the bus
- Connection: a physical connection which is bound to the user
- Channel: a logical stateful connection which is tied with the physical connection
- Exchanges: entities to which the messages are sent
- Queues: entities which receive messages
- Messages: the actual message sent to any exchange
- Bindings: relationship between an exchange and a queue

2.4 Choice of Web server

The choice of web server was not central to our project. We did not really need a lot of speed to our project and it seemed in our minds that we would not be doing a great deal of work related to the web server. Still, a choice had to be made, and we filtered down the choices to select few during the first weeks. These choices were intimately tied to the choice of web framework since the prospect of implementing support for a new web server for any web framework was not very attractive.

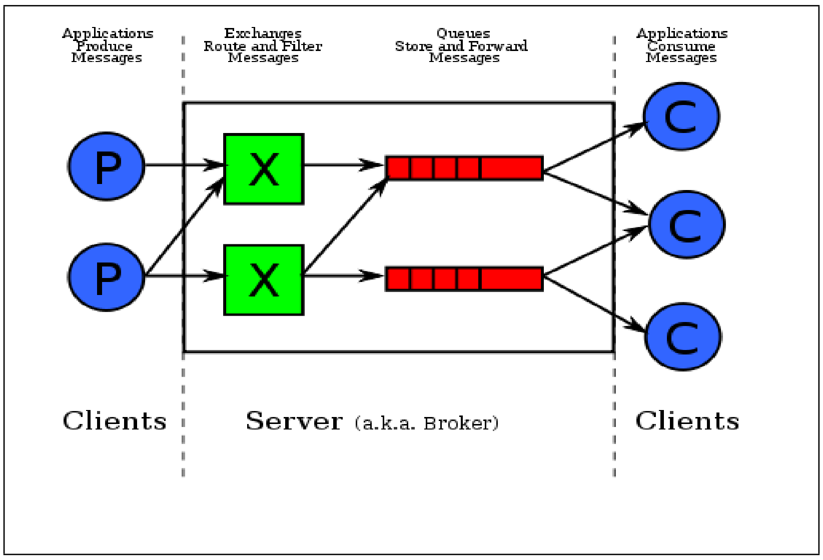
Applications Produce Messages
Exchanges Route and Filter Messages
Queues Store and Forward Messages
Applications Consume Messages
P
P
X
X
C
C
C
Clients
Server (a.k.a. Broker)
Clients

Figure 1: Two clients connected by an AMQP broker.

2.4.1 Yaws

A main contender from the start, Yaws [5] is a lightweight web server designed for speed and parallellism, something which is always popular with Erlang programmers. Its age meant that it was rather well documented and, more importantly, it worked relatively easy on its own. However, we did have the problem of Yaws being over-active and hosting it's own default web page when we expected Erlang Web's 2.5.1 or something else. This turned out to be Ubuntu's fault, since when you install Yaws in Ubuntu [16], it adds a startup script that starts Yaws in the system upon startup, which is rather strange.

2.4.2 Mochiweb

Although described by many as being "neat" and "cool", we found the documentation of Mochiweb [20] to be fatally lacking. It describes itself in places as a library for writing web servers, and felt unintuitive to use overall. It did work well with Nitrogen 2.5.2 (having been developed by the same group), but when we started leaning towards Erlang Web 2.5.1 it fell to the wayside.

2.4.3 Inets

This module [21] is a part of Erlang 2.1 provides the most basic API to the clients and servers, that are part of the Inets application, such as start and stop. Though it is supported by both Erlang Web 2.5.1 and Nitrogen 2.5.2, it is apparently rather basic, and we simply preferred Yaws 2.4.1 over it.

In the end we chose Yaws as our web server of choice, and it has worked well for us.

2.5 Choice of Web framework

Since the actual renderer of HTML [22] is a central part of a website solution, the choice of application that we would use of course needed some deliberation. After discussing and researching many options, we came up with a list of candidates which were more or less fitting.

2.5.1 Erlang Web

Perhaps an obvious choice, considering Erlang Solutions [15] part in developing it and has extensive experience using it, but we still started out unsure as to whether or not it would be the best choice for our system. Erlang Web [6] seemed to be designed for professional use and not for the layman. This is evidenced for example it's large amounts of modules and a seemingly high cost of entry to actually make a webpage. The application has also not been updated (that we could gather) for quite some time, which was quite different from the other solutions we looked at which were all quite alive and had recently released updates and patches.

With Erlang Solutions investment in Erlang Web, it also felt natural to choose it since we could presumably get support if we ran into problems with it. In the end we did not need very much support regarding Erlang Web, it is fairly straightforward in its construction and structure, but it was still felt like a large incentive for us in favor of Erlang Web.

The promise of a fully fledged CMS (Content Management System)3.10 was also alluring in the decision making, since we initially thought that we would be developing it ourselves if we were not able to find a reasonable one to use (and given the apparent lack of web applications written in erlang we did not feel too optimistic about it). An already finished CMS would save us a lot of time and effort then, and hopes were that we would be able to get it off the ground quickly and easily.

Erlang Web also uses the very common MVC (Model, View, Controller) model [23] of web applications. We envisioned this as making our re-modeling of the framework into using our application easier, since you keep the logic (the controllers) in one place, and the actual data in other places. Since Erlang Web uses templating, storing the static HTML-content in a separate place from the logic that produces the dynamic content, we could save quite a bit of traffic over the bus. We also hoped that Erlang Web would have a good separation of different modules so that we would be able to switch out the regular components it had for our own versions that use the system we'd construct.

2.5.2 Nitrogen

The framework of choice for most Erlang Web 2.5.1 developers it seems, Nitrogen [12] initially wooed us with it's fancy Ajax (shorthand for Asynchronous JavaScript and XML) [24] incorporation. Making a webpage in Nitrogen is also quite simple, and allows you to write rather small amount of Erlang 2.1 code to make pages. Muchlike Erlang Web 2.5.1, Nitrogen also has rather nice layering of abstractions, although the codebase is rather hard to penetrate. We found the documentation to be lacking and it seemed rather hard to make the company-like webpage we were aiming for in our project. The event-driven nature of Nitrogen seemed very nice, but in the end the MVC (Model, View, Controller) model [23] felt more comfortable to us. Considering how you construct webpages in Nitrogen (constructing records containing the text and formatting for individual elements and drawing them), we felt that Erlang Web's 2.5.1 approach was more clean, and allowed us to distribute the work more easily. The folder and module structure was hardly documented in Nitrogen so changing its behaviour would probably have been more work as we would have needed more time to investigate the source code and the structure of the application before we could start changing it.

Another key problem that Nitrogen had that we could not fix was that it did not run with Yaws 2.4.1, which we at that point, had already decided to use. This was big disadvantage for Nitrogen.

2.5.3 Chicago Boss

Several alternative Web frameworks are being worked on actively, and one of the more interesting is Chicago Boss [25]. Unfortunately it's more in pre-alpha stage than production ready. It was never a very serious candidate but is worth mentioning for being interesting. From reading the code and following the mailing-list we soon concluded that it's poorly documented and has quite a bit of bugs.

```
<?xml version="1.0" encoding="UTF-8" ?>
<rss version="2.0">
  <channel>
     <title>Liftoff News</title>
     <link>http://liftoff.msfc.nasa.gov/</link>
     <description>Liftoff to Space Exploration.</description>
     <item>
       <title>Star City</title>
       <link>
          http://liftoff.msfc.nasa.gov/news/2003/news-starcity.asp
       </link>
       <description>This is the description </description>
     </item>
   </channel>
  </rss>
```

Figure 2: An example RSS Feed

2.5.4 Zotonic

An interesting candidate, Zotonic [13] is a fairly polished web framework with a built-in CMS (Content Management System) 3.10. Unfortunately Zotonic only runs on PostgreSQL (SQL type relational database) [26], a very complex database which is not written in Erlang 2.1. Altering Zotonic to use CouchDB 2.2.2 would have taken a lot of time (although it is hard to say if it would have saved us time in the end since it has a lot of features), but since we considered the matter of the CMS to be secondary in our project, and in the end started working on it only in the final few weeks, this decision was probably for the best.

2.6 RSS

RSS (Really Simple Syndication) [27] is an XML [28] format for sharing frequent updates between web sites. RSS documents known as RSS feeds allow webmasters to syndicate web content automatically. They allow readers to quickly check for news and updates from favored websites. An example can be seen in the figure 2.

RSS feeds are written in XML. They begin with the XML declaration followed by the RSS document type declaration. All elements are surrounded by matching start and end tags. These elements are case sensitive and must be properly nested. The values of attributes of each element must be quoted. This turned out to be a problem because an RSS-parser that parses correct feeds will fail to parse most of the actual RSS feeds available online.

2.6.1 Ibrowse

Ibrowse [29] is an application implemented as an HTTP [22] client in Erlang 2.1. The Ibrowse module has a basic function send_req which takes 3 to 6 arguments and sends the HTTP request to the supplied URL and gets the reply. The basic syntax is
browse:send_req("http://www.google.com/", [], get)

We have used Ibrowse in the implementation of the RSS refsec:rss feeds where the request is sent to the RSS subscriber and displayed in our site.

2.7 Rebar

Rebar [30] is an Erlang 2.1 build tool which we used for easy compilation, testing of our applications and handling releases.

Rebar is written in Erlang. Also, rebar uses standard Erlang/OTP conventions for project structures. Rebar provided support for most of our development such as compilation, EUnit 5.1 and providing it's coverage analysis, and document generation through Edoc 2.10.

2.7.1 Conventions to be followed for rebar to work properly

The application directory should follow the OTP standard, and so we do, with a few addendums to accommodate other functionalities.

- test : where the test files are kept. EUnit automatically looks for this directory for test files.
- c_src : where C source files are kept. Never actually used in this application.

2.8 Project Management - Redmine

Redmine [7] is a project administration tool useful for managing projects. It has lots of documentation and plug-ins like a scrum dashboard, burndown charts, backlogs, forums, e-mail clients etc. Scrum dashboard was used to keep track of project progress according to our team methodology. Burndown charts is graphical representation of work left to do. The outstanding work is on the vertical axis and time on the horizontal. The chart starts at the left up corner and ideally should end up in the right down one. Backlog is a histori of a project. Forum provides you with opportunity to communicate with project members on-line.

2.9 Version Control System - Git

Version control systems are a major part of any project. It is used to keep track of the code of that has been written. Git [31] was made and popularised by Linus Torvalds and is generally considered to be one of the better Version Control system. We had some issues using it but for the most part it worked well for us.

2.10 EDoc

Writing a good program is not the only responsibility as a programmer, one has to document all the code so that it can be used as the future reference for someone new to understand them. In our project, we used EDoc [32], an Erlang 2.1 program documentation generator.

EDoc is an Erlang program that automatically generates the documentation written inside the Erlang modules as comments in a fairly straightforward manner in the HTML [22] format. There are certain rules for writing the comments in the program and if one follows that rule then EDoc does the rest. There are special tags such as "@Name,@Doc, ..." which are used while commenting.

There are many different ways to run the Edoc but since we use Rebar 2.7 as the building tool, it gives us functionality to generate the docs through the command "rebar doc" which will parse the source code, generate and organize the documentation in the "doc" folder.

Almost all of our source code is documented using Edoc. By generating the documentation through it you can read through what most of the modules do fairly quickly and get a feel for the applications.

Unfortunately Edoc does not put all of the documentation in one place but everything is spread in different folders. We had problem of documenting the test files using Edoc, as it usually goes for the "src" folder for the files to be documented rather than any other folder. Another disadvantage of using Edoc was that no functionality was provided to document functions with several function clauses. It ignores them and documents only the first function.

2.11 Meck and EUnit

For unit testing, we used the EUnit 5.1 library to construct unit tests for our modules. It allows us to create testing files where we can test the functionality of a module by calling its functions with predefined parameters and expecting certain results. EUnit allows for efficient use of a multicore system by parallelising the tests. But it still gives you control to order the tests. Many of our tests have order enforced due to the importance of side effects while testing a concurrent system.

While EUnit is sufficient for creating and managing our unit tests, the distributed nature of the project made unit testing hard. To solve this problem we used a library called Meck.

Meck [2] was written by Adam Lindberg who works at Erlang Solutions. Meck is a library that is used for spoofing function calls. It can be used on any function call from a module that is not a sticky module.

With Meck, you can select a function in a module and simply assert a new function to that call. Every time the original function is called, Meck will make sure that the substitionary function will be executed in its stead. A simple example can be seen in figure 3. This functionality is used to spoof most parts of the system during our unit testing.

```
1> meck:new(dog).
ok
2> meck:expect(dog, bark, fun() -> "Woof!" end).
ok
3> dog:bark().
"Woof!"
4>
```

Figure 3: A simple Meck example

2.12 Hardware and OS

We were mainly using nine 2.66GHz Intel Q9400 computers running with 3GB RAM built by HP loaned to us from Uppsala University [33] during this project. In addition we use three dual core machines of a lesser capacity. All stress testing has been performed on the quad core machines.

The operating system we have been running during the development is Ubuntu 10.x [16], updates to the OS have been applied as they have become available.

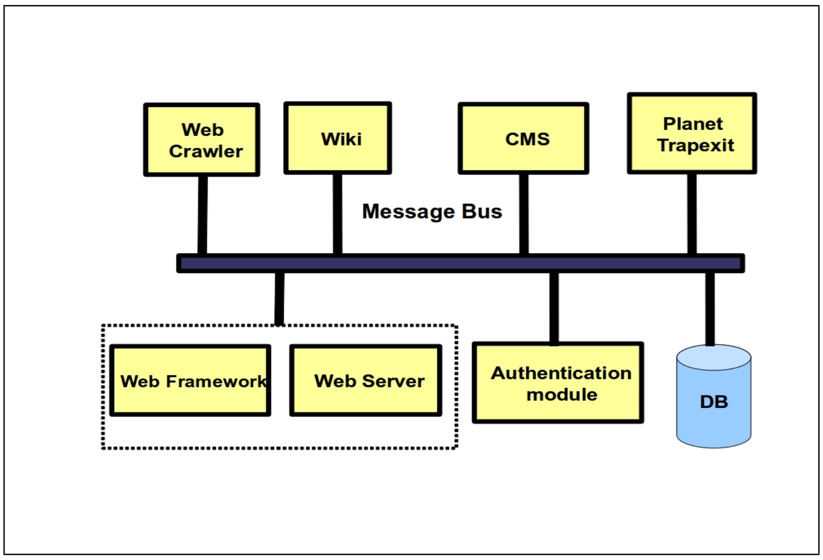

Figure 4: An overview of our system

3 System design/architecture

The previous sections described what tools we used to build our system. Now we will go into more depth, detail our system design and explain how we use those tools to achieve our goals.

3.1 System overview

It provides the overall view of our system. The architecture seen in the figure 4 was a suggestion, given to us by Erlang solutions [15] as a structure they wanted to be implemented. Throughout the project we've come to implement and test the result described in this chapter.

We have built the entire project upon a service structure where we have a message bus, database, web framework, web server, content management module and authentication module as backbone. On top of the backbone we can then add services to the system such as the planet trapexit (which is the start page of the website, displaying news etc.), wiki and forums.

3.2 Message Bus

In our implementation, RabbitMQ 2.3 is the core of the system which connects various components in the system such as the database 3.4, authentication module 3.5, CMS (Content Management System) 3.10 . . . No components are directly connected to each other. All the requests between components are routed through the message bus. The message bus acts as a mailman which gets requests from components and delivers them to other components in the system.

In our system, the RabbitMQ is a standalone application and all the services or components needs to establish a connection with it to be a part of the system. At first, services need to be registered with the RabbitMQ with proper username, password and privileges. We also have an API for the RabbitMQ which is used by all the components to connect to the bus. When the service requests for the connection with the message queue with valid credentials, the following things occur:

- A connection is established
- A channel is opened
- An exchange is created with name as username_x

- A queue is created with name as username_q
- The exchange and queue are bound together (meaning messages to the exchange go to the queue automatically)
- The component is subscribed to the queue if it is specified

The user can choose to either subscribe its queue or get the message when required. If the user chooses to subscribe to the queue, the messages are pushed to the user as soon as they end up in their queue otherwise they will have to get it manually from their queue one at a time or all at once. Subscribing is recommended if you intend to send and receive many messages, since polling always has been slow as a method for message distribution.

All the requests and responses in the system are translated to messages while sending and reverted back when receiving.

There is a comprehensive list of API calls in the User Manual appendixB.1.

3.3 Message Structure

We came to the conclusion after several discussions that the messages that we pass between the services should be fairly simple.

Simplicity was our aim in designing the message structure. We have used records for this, as it would be simple and general to send messages to/from the connectors 3.8.1 to services. Our message record format is more general and the actual data is encapsulated in the payload. In this way we abstract the data.

```
#message{sender:string(),priority:int(),payload:any(),id:int(),options:list()}

Sender: Sender of the message
Priority: Priority of the message (Not currently used)
Payload: The payload
Id: Used for keeping track of requests.
Options: Currently unused. May well be removed
```

3.4 Database

As detailed in the tools section, the current database integrated in the system is CouchDB 2.2.2. CouchDB is a document oriented database and it not supposed to have any structure or schema in the general meaning. But one can be imposed upon CouchDB via an API.

3.4.1 Schema architecture

In CouchDB 2.2.2 data is stored in documents and databases. To separate data which is logically not suppose to be stored at one place we use different databases. Each database in CouchDB is a named data space which in SQL corresponds to a table. So we will use that meaning of "database" in the report. For example, to register users we have to store many different values: user ID, the user's personal information, password etc. So each document in a database contains several fields. A document corresponds to a record in an SQL table. That is example of a simple (flat) data architecture. At the same time there might be services with more complicated structure.

3.4.2 Views in CouchDB

A view makes it possible to get a number of documents from one database in one query. Unfortunately, it is impossible to link several databases in one view. To implement a new view in CouchDB 2.2.2 that will work with our APIs, you have to follow these rules:

- there should be one (and only one) design document in the database.
  The name of the design document must be "_design/X" where X is the name of the database (e.g."_design/users") .
- All views should return list of lists of field_name and value: [[field_name, value], ...].
- Each view has its own unique name. An example of the view syntax can be seen in 5.

```
if(doc.Shoesize && doc.Name){
    emit(doc.Shoesize, [["Shoesize", doc.Shoesize],
                        ["Name",doc.Name]]);
    }
```

Figure 5: The first parameter in the emit function is the key and the second is the value to be returned.

3.4.3 Database payload structure

In the database API, we construct payload by making a record. The below snippet of code gives the format for the incoming messages to the database.

```
#payload{command:string(),storage:string(),id:int(),data_value:list(),options:list()}

command    : Type of database command that is to be executed,
             they are: "read", "update", "insert", "delete".
storage    : The name of the database,
             for example: "wiki", "forums", "thread", "posts", "web_page" "users"
id         : Identifier of the object to be read or the view to use.
data_value : Is relevant in INSERT and UPDATE operations.
             Format: data_value: [{field_name, <value>},...]
options    : Not used.
```

3.5 Authentication

We have implemented the authentication as a separate service which is connected to the message bus. So, whenever some different services need to interact with the authentication service, they can simply use the authentication API and the messages will be routed over the message bus.

Our implementation of the authentication include:

- Encryption of passwords. We have used the MD5 hash function for encrypting the password but before the encryption, a 4-digit random number salt is generated and then appended to the end of the original password and stored in the database.
- Verifying login attempts, controlling whether or not an entered password is correct.
- Registering users.
- Changing the password.

The authentication module needs to communicate with the database to store the user's information. Service requries "users" database to be created in the CouchDB 2.2.2 before it can insert the user. So, it is the responsibility of the authentication service to ensure that the users database exist beforehand and this should be done manually. Whenever authentication service needs to communicate with the database, it will use the external database API provided by the database for all the database specific operations and whatever is done beyond that is totally specific to the database only. But a confirmation regarding the operation is delivered to the authentication service.

3.6 The web server module

The part of the system which is perhaps the simplest, our code simply starts an instance of yaws 2.4.1 with the "ws_callback" as callback module. It passes along requests that the web framework can answer and expects responses from it.

In the beginning of the project we expected the web server to run on a different node than the web framework, and made an implementation that made them run on different nodes. Later on we decided to change this for efficiency reasons. We made new connectors 3.8.1 to make this work, essentially bypassing the message queue but keeping the logical separation. It is interesting to note that if you wish to change the two components back to communicating with the message queue it would be rather trivial. Just change the connector that is used to a gen_connector B.5 instead of the specialised ws_connector. Nothing needs to be changed in the web framework since it handles the web server requests in its callback module.

3.7 Web Framework

The web framework is a central part of our system, but rather hard to categorise. Over the project course it has done different things and to make a comprehensive list of tasks that it performs is hard. Erlang Web 2.5.1 has many features that we do not really use, and these are not documented at all in this report. The things that we do use are documented here in a "black box"-fashion. We don't really know for sure how Erlang Web does many of the things it does, we just assume it does them correctly (as it should).

The things we have the web framework do for us is mainly:

- Dispatching the different requests to different controllers
- Requesting templates and data sets from the controllers that the dispatcher contains over the message bus.
- Expanding templates and inserts data from controllers into it.
- Handling sessions
- Communicating with the web server, completing requests from it.

3.7.1 Division of labour

The division of labour between the web framework and the services was something that was discussed at length in our project. In the end, we settled on an approach that is basically a bit of a compromise. The web framework will have it's own dispatcher to determine "where" a request should be handled, pass the relevant state (session, path, the request itself etc.) to the controller and wait for a reply. The controller will set the relevant dynamic parts of the web page and return the name of the template to be used, making database queries. The web framework then expands the template and inserts the dynamic data supplied by the controller.

3.7.2 Dispatching

We use the Erlang Web 2.5.1 default dispatcher, which allows you to specify regular expressions in a configuration file to dispatch incoming requests based on the URL.

```
logged_in  : status flag
https      : status flag
controller : tuple with module, function and arguments
post       : POST data
get        : GET data
path       : URL
cookies    : list of cookies
cookie_key : session cookie value
ip         : IP address
session    : session state
```

Figure 6: #request record (web framework → service)

```
template   : name of/path to template
data       : dynamic data
```

Figure 7: #response record (service → webframework)

3.7.3 Session Handling

We use e_session, which is provided out of the box from Erlang Web 2.5.1. An API has been constructed for services to access the session indirectly. For details, see the User manual appendix, section B.2

3.7.4 Changes made to Erlang Web

As we initially ran the web framework and the web server on different nodes we had to make a small change to Erlang Web's 2.5.1 start script to stop it from starting its own server. Though we put two components back on the same node we still kept them logically seperated and thus needed to keep the change as well.

3.7.5 Service Protocol Structure

As the webframework requests a page from a service it sends an instance of a record (see 6) defined in the webframework API header 'wf_api.hrl'. This header also defines the record (see 7) used for the reply sent from the service to the webframework.

3.8 Double Abstraction

The system architecture(as seen in figure 8) is subject to what we call a "double abstraction". Speaking in terms of design patterns this is also known as the bridge pattern.

Initially, we figured that we wanted to abstract away the message queue from the services point of view. The diagram 8 shows this abstraction. None of the components are communicating directly, all of the data is transfered via public API's and through our module "gen_connector".

The result of this is that the components are all exchangeable. Even the message queue itself is exchangeable as it is abstracted away from all the other components via their connectors.

3.8.1 Connectors

In the beginning we had a lot of discussions about how to implement the double abstraction. We have achieved double abstraction by implementing connectors for all of the components. We

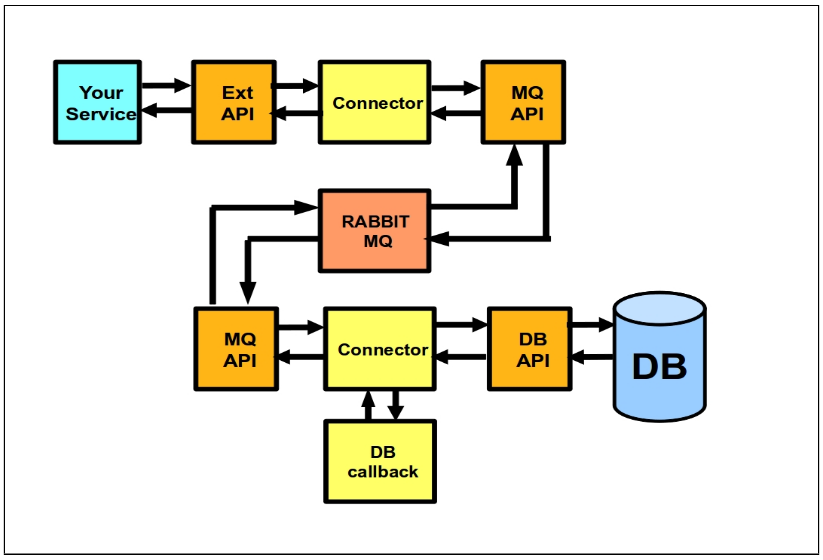

Figure 8: A better view of double abstraction of our system.

simply let the connectors handle all the communication between them and the message queue.

The connector is the main communication module in any given service. The module is called "gen_connector" and it requires the name of a callback module as its starting parameter.

This callback module contains instructions on how incoming messages should be handled. Then when it receives a message, the connector spawns a new worker child that calls the callback module with the contents of the message. The result is sent back to the gen_connector who will reply with the result to the sender.

The connector is also used when a service wants to send outgoing messages. Calling the "gen_connector:send/3" function will cause the gen_connector to send a message over the message queue to the designated recipient. The callback module in turn uses our RabbitMQ 2.3 API for communicating with the message queue. This API serve the purpose of abstracting away the message queue from the connector. The connector also keeps an internal record of messages that are awaiting replies so that it can match the correct replies to the original message.

The gen_connector will also upon startup read from a configuration file called [service-name].cfg. (e.g. db.cfg) where it will extract the login name and password for the message queue so it can establish a connection with it. If this operation fails, connector startup fails.

Note that functions for making synchronous calls are implemented here, not in the MQ (Message Queue). This was something that simply happened, and it should probably be moved into the MQ (Message Queue) api instead, but we never found time to do it.

3.9 RSS

In our implementation, the URLs to the RSS [27] feeds are stored in the database. The first step is to create a database for RSS. In our case, there is a script (i.e. inst_db_schema) which performs the following:

1. create a database called 'rss'

2. create a view called 'feeds'

The 'rss' database should contain documents containing URL field. After that, all the stored URLs from the database are retrieved. Since CouchDB 2.2.2 is used as the database, a view needs to be created to extract all the URLs. To read RSS feeds, an RSS reader (or aggregator) is implemented. In the RSS reader, the URLs are passed to a function in ibrowse 2.6.1 module, an HTTP client written in Erlang that provides extensive HTTP support, which returns the RSS feeds in XML format. The retrieved XML feeds are then parsed using xmerl, an Erlang library. The parsing results are grouped into a list of list of tuples, where each list of tuple is a single RSS feed. The results are then sent to the web framework for display.

3.10 The CMS (Content Management System)

Although shipped to us at a very late stage of development and not being a focal point of this project, quite a bit of work has been done on the CMS [35] to integrate it into our system.

Currently, the CMS is in a semi-working state, wherein you can run and view some parts of it but it has some glaring problems.

1. Many parts of the CMS don't actually work and none of it has been tested methodically.
2. There is no way to automatically set up the database properly. If you wish to use the CMS you will have to set up the CouchDB 2.2.2 instance to have the proper, expected databases, views and documents to make it work.
3. It isn't connected to any public website. You can alter things in the database (assuming it is set up properly), but the public part isn't complete yet.

The CMS is an application built on top of Erlang Web 2.5.1, which may seem like a good thing when you want to integrate it into our system. It certainly seems more convenient than converting a Nitrogen application, but the fact is that since we do not expose the features of Erlang Web 2.5.1 directly, it became very hard to insert the CMS into our system. Erlang Web 2.5.1 is very cross-referencing between the different parts that we have separated. Therefore, since we wanted to keep the different parts of the systems separated from each other, it is troublesome to change all the calls between the two "areas" where things are kept.

To deal with this, we had to implement some wrappers for different functionalities that the CMS expects (or any Erlang Web 2.5.1 application expects). The two main modules we implemented to cope with this are backends to e_auth and e_db, aptly named "e_auth_trapexit" and " e_db_trapexit". They can be found in the ext_api directory and implement functions that, opaquely, make Erlang Web 2.5.1 applications use the message queue. They implement the same API as e_auth and e_db, and can be used as the backend of an actual Erlang Web 2.5.1 application running as well. Note, however, that we've changed all the function calls in the CMS to actually call to the e_db_trapexit module, since we do not start the applications that set up the state that makes e_db work in the same node as the controllers. We also use some Erlang Web 2.5.1 modules directly on the controllers sides of the system. For example, we set up an e_dict on the node that the public and cms are running so that their calls to wpart:fset and wpart:fget work properly.

In order to set up the database to work with the CMS, you need to:

- Create databases with names corresponding to all the wparts in the cms application, admin_panel and so forth.
- Create a document called "meta" with a field called "counter" in it. It should be set to some integer value like 0. It is used for creating ID's for new elements of the corresponding type.
- You need to set up a view for all of the new databases that returns the ID of all elements and their data fields. For an example of such a view, see figure 9.

```
function(doc){
    if(doc.data)
        emit(doc._id, [[doc._id, doc.data]]);
    }
```

Figure 9: An example view that can be used to set up the CMS

3.10.1 Dynamic Menu

In the ‘public’ part of CMS, a dynamic menu was created. Due to lack of time, the dynamic menu was not fully implemented as the way we wanted it to be. Currently they were created with different database schemas with respect to the ‘admin’ database schema. Therefore, if a menu is created in the ‘admin’ part of CMS, they will not be viewable in the public site.

How does the current dynamic menu work for us? The ‘public’ site could read information about the menu and their entries from the database and display them.

The names of the menus and their entries are stored in different databases, named ‘cms_menu’ and ‘cms_entry’ separately. The database schema for menu is flat in the sense that there is no reference to other databases. The database schema for the entries contains relations to the menu database. Each entry document has a menu_id refering to what menu it belongs to.

All templates are loaded when the CMS is started. In the HTML page, wpart tags are used to refer to the controller of the core menu with a target attribute specifying which site (public or admin) to load. Inside the controller of the core menu, the names of the menus and their entries are retrieved from the database and converted to a record of type ‘core_menu’ and ‘core_menu_entry’. The record is then applied to the predefined template. The final template is sent back to the web framework for display.

3.11 Planet

The planet is an application that we made before gaining access to the CMS application. It runs properly in our system and uses Erlang Web’s 2.5.1 templating. It is not connected to the CMS however and can basically be considered to be a test-application that shows the system working in full.

3.12 Applications and Supervisors

Our project has many applications and each component has to be a stand-alone application, so that they can be easily exchangeable. A brief explanation of our application structure can be seen in figure 10.

Applications were used so that it would give us a nice binding for all our API modules and supervisors were used to make our system fault-tolerent. The supervisor starts its respective child processes. The child process is the gen_connector 3.8.1 module for all our applications. It then gets connected to the message queue by giving its credentials. This works the same for all the services and components except the web framework and web server.

The web framework supervisor starts two child processes. The first one is the gen_connector module which works the same way as it does for the others. Second child is the wf_bridge module. The web server supervisor starts a separate connector called ws_connector. Because in our system, the web framework and web server are joined together and are not separate applications. So the web server communicates directly with the web framework instead of via the message bus.

For all our supervisors, we have used the “one-to-one” restart strategy.

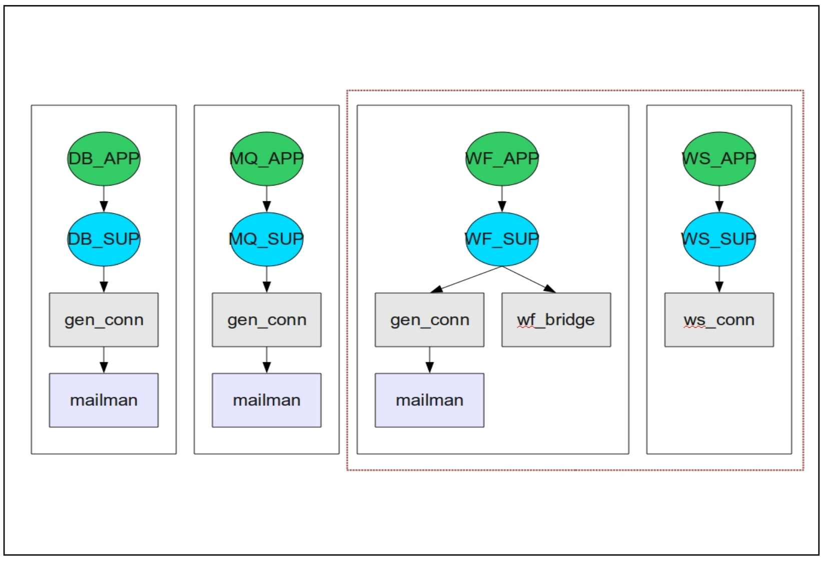

Figure 10: Example application and supervisor structures

4 Dataflow

This section describes the dataflow of our system, in terms of how a simple transaction between two services work, how HTTP [22] requests are processed, and how an end user manages his account on Planet Trapexit.

4.1 A Simple Transaction

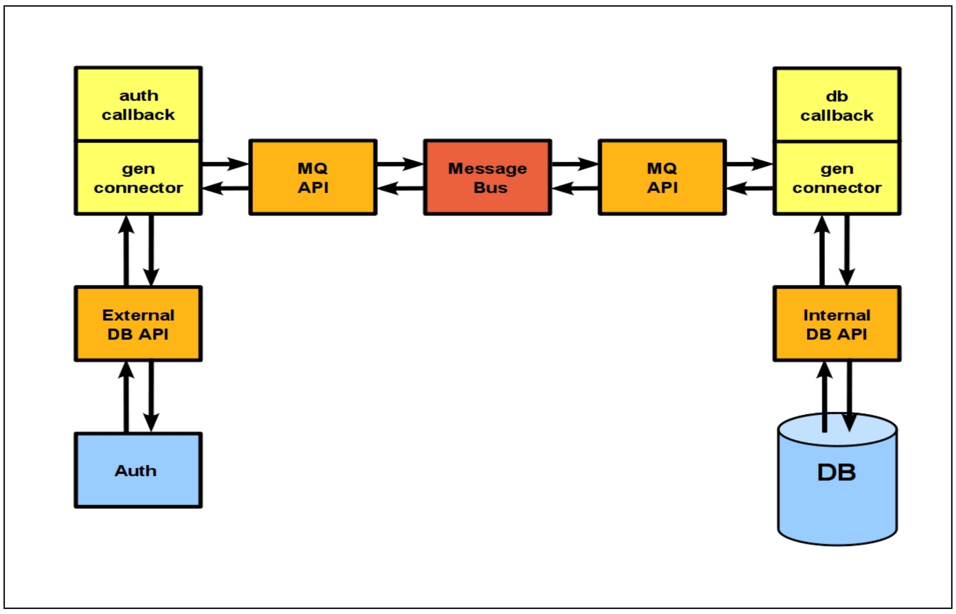

Figure 11: In order to get information from the database, the authentication module must communicate via the external database API which is available for all services.

The figure 11 depicts a simple transaction between the authentication service and the database service. Before starting the communication, the connectors must be started on different nodes. This will ensure that these two modules are connected to the message bus. The external database API will send a message to the message bus. When sending a request to the connector, the connector will package the payload(i.e. the actual request) into a message record before sending it. The message record is again packaged into AMQP (Advanced Message Queuing Protocol) [11] record by the RabbitMQ 2.3 API's process before it is sent to other service. The message is received by the connector at the database end and the actual request is passed to the database callback module.

After getting the request, the database callback module will call the internal database API which is private to the database service itself. The internal database API will query the database according to the request. The results will be sent back to the authentication module through the message bus in a way that is analogous to the authentication to the database transmission. The RabbitMQ 2.3 API's process at the receiver's end will unwrap the AMQP (Advanced Message Queuing Protocol) [11] message and deliver it to the authentication's connector which again unwraps the message record and extracts out the actual payload of the message.

4.2 Handling HTTP Request and HTML generation

When a user requests for a page, the steps followed are

- At first, the request in the form of an URL is sent to the web server (Yaws 2.4.1)

- Yaws then calls the 'arg_rewrite' function in the web server callback module which lets us edit the request before it is handled
- The web framework is queried with the URL and replies whether or not it refers to a static object. This allows us to serve static objects such as images directly from the web server
- If the request is not static a prefix is added to the URL which causes Yaws to call the 'out' function in the callback module to handle the request
- Since the web server and web framework are on the same node, both components have a special module for communicating with each other: ws_connector and wf_bridge respectively. The web server callback module thus calls the wf_bridge via the ws_connector
- The bridge will then spawn a process for the web framework callback module to handle this request. The web framework callback module sets things up for Erlang Web 2.5.1 and figures out what to do with the request, i.e. what service to call over the bus, and the request is sent to said service using the gen_connector
- The services return the name of a template to use along with relevent dynamic data, and Erlang Web 2.5.1 is used to combine these two into the final HTML [22] which is sent to the web server.

4.3 User Management

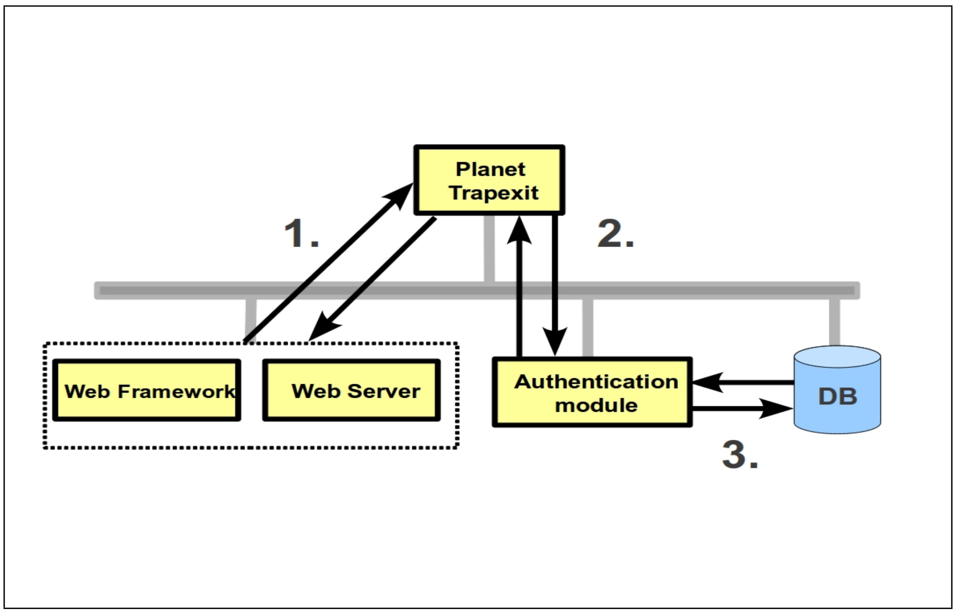

Figure 12: An HTML Dataflow

4.3.1 User Registration

When an user registers, the data will be processed as follows:

- the web browser sends user's data to web server.
- the web server receives the data and sends it to the Web Framework via Web Framework bridge.

- the Web Framework receives the data and sends it to the Planet Erlang service.
- once the planet receives the data, it calls the authentication module to insert the user's information into the database.
- the authentication module encrypts the password and calls the external database API to add the user in the database.
- the database will reply 'OK' if the user is inserted successfully, or an error message is returned if the user already exists in the database.

4.3.2 User Login

When an user logs in, the data will be processed as follows:

- The first three steps are the same as in user registration.
- the planet component calls the authentication module to check if the user exists.
- the authentication module calls the external database API to read the user's information from the database.
- the database will reply 'OK' if the user exists or send back an error message.

5 Testing

Testing plays one of the most important roles in the IT development process. In this section we will cover how our tests are structured.

5.1 EUnit

5.1.1 How to do the test

Test cases in EUnit [34] is a stand-alone Erlang 2.1 module where developer calls tested functions and matches its return values with predefined correct values. It means that developer can not write one test cases module for more than one project modules. Name of the test cases module should be the name of the module followed by "_tests.erl".

5.1.2 Result

If a test is done successfully then the test report will contain 'Test is done, ok'. If one or more tested functions returns a value that differs from the expected result then the test is considered to have failed and a report will be generated. Using EUnit in Rebar 2.7 also gives you detailed code coverage report where you can see how much code is executed by the test suite.

5.2 Tsung testing

Tsung [8] is a tool built to test the scalability and performance of the client/server applications. It is written in Erlang 2.1 and is used for stress testing the servers. When testing the system, the system is setup in some different ways for comparison reasons. For all tests the Tsung testing suite was run on one node.

The setups can be seen in table 1 and the different results in figures 13, 14 and 15.

Setup #	Node 1	node 2	node 3	node 4
1	Database Message bus Webserver Webframework Authentication Module Planet Trapexit			
2	Database Authentication Module Planet Trapexit	Message bus	Webserver Webframework	
3	Database Authentication Module	Message bus	Webserver Webframework	Planet Trapexit

Table 1: Tsung test setups

- Figure 13 displays the results for the test run where the whole system is running on one node. The results show that the system performs well up to about 210 login attempts per second.
- Figure 14 show the graph for the test with the Web Framework 2.5.1 and the Web Server 2.4.1 are running on their own nodes. The graph shows some significant improvement

from the all in one physical node test and the system can now handle a bit over 270 login attempts per second.

- Figure 15 show the graph where the web page also run on a separate node and the performance is pushed a little bit higher and the system handles around 300 requests per second.

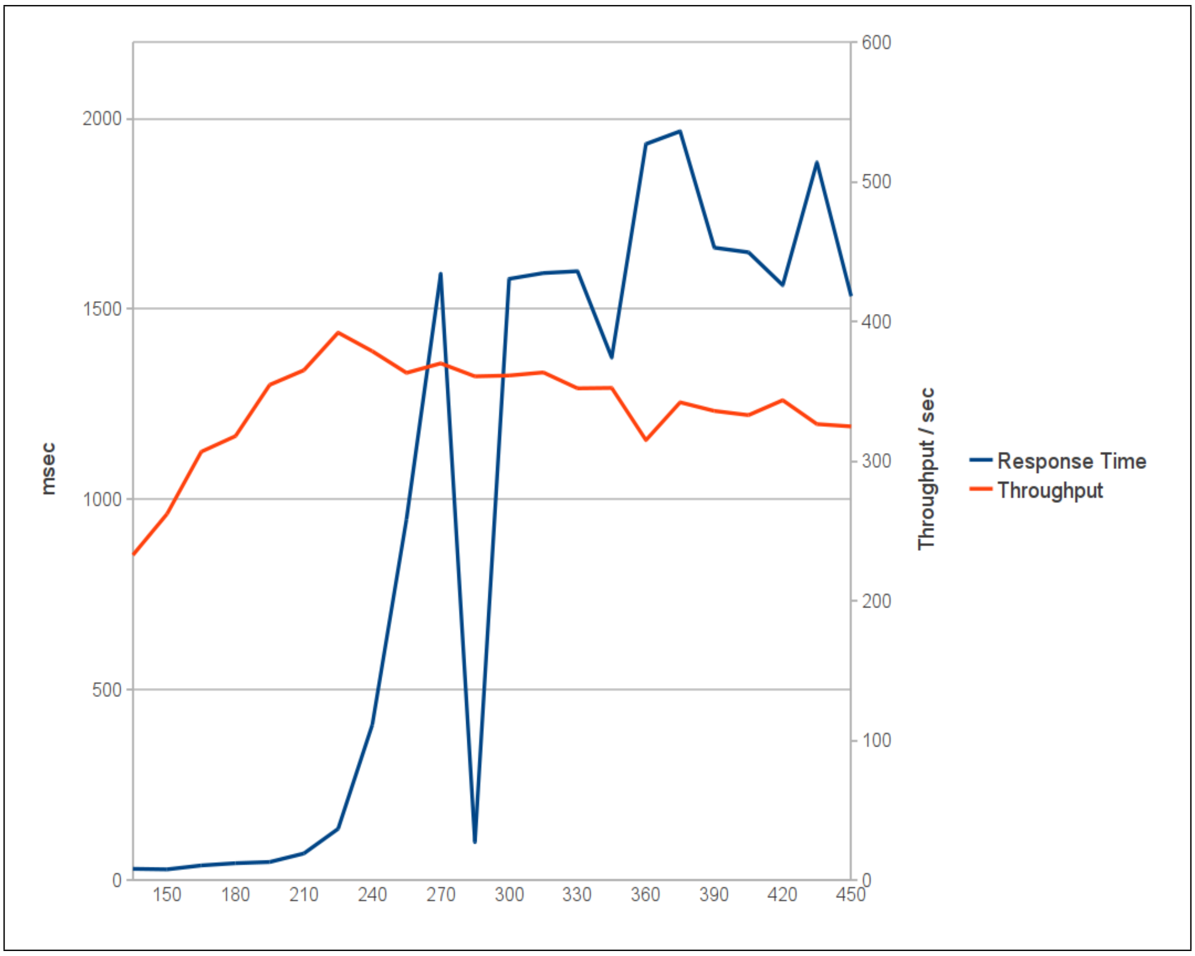

Figure 13: All in one node

5.3 Extracting Tsung Data

The Tsung [8] data we were interested in was the average response time and the Transactions per second. In the HTML report that Tsung generates, these graphs are represented as images called graphes-Transactions-mean.ps and graphes-Transactions-rate.ps. To find out the corresponding data we checked the gnuplot files that generate these images. They are available in the log-folder/gnuplot_scripts/graphes-Transactions.gplot

When you open up this file, you will see the gnuplot instructions for generating the above images. To generate graphes-Transactions-mean, gnuplot reads column 1 and column 3 from a file called page.txt. Likewise, to generate graphes-Transactions-rate gnuplot reads column 1 and column 2 from the same file, page.txt.

Opening up this page.txt file (available in log-folder/data/page.txt) gives you access to the data. It was then pulled out manually into Open Office [38] where we made more elegant graphs out of it.

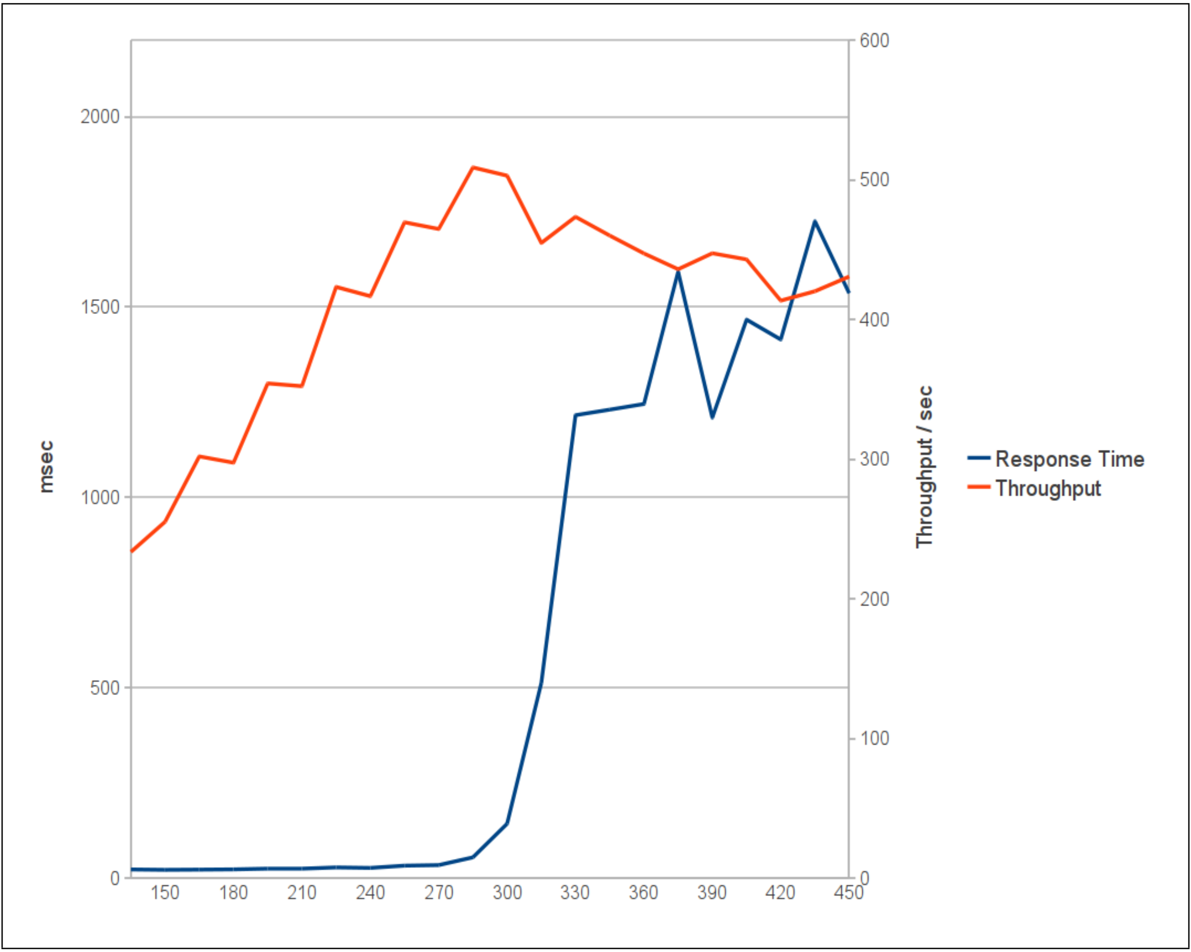

Figure 14: Web Framework and Message Bus separated

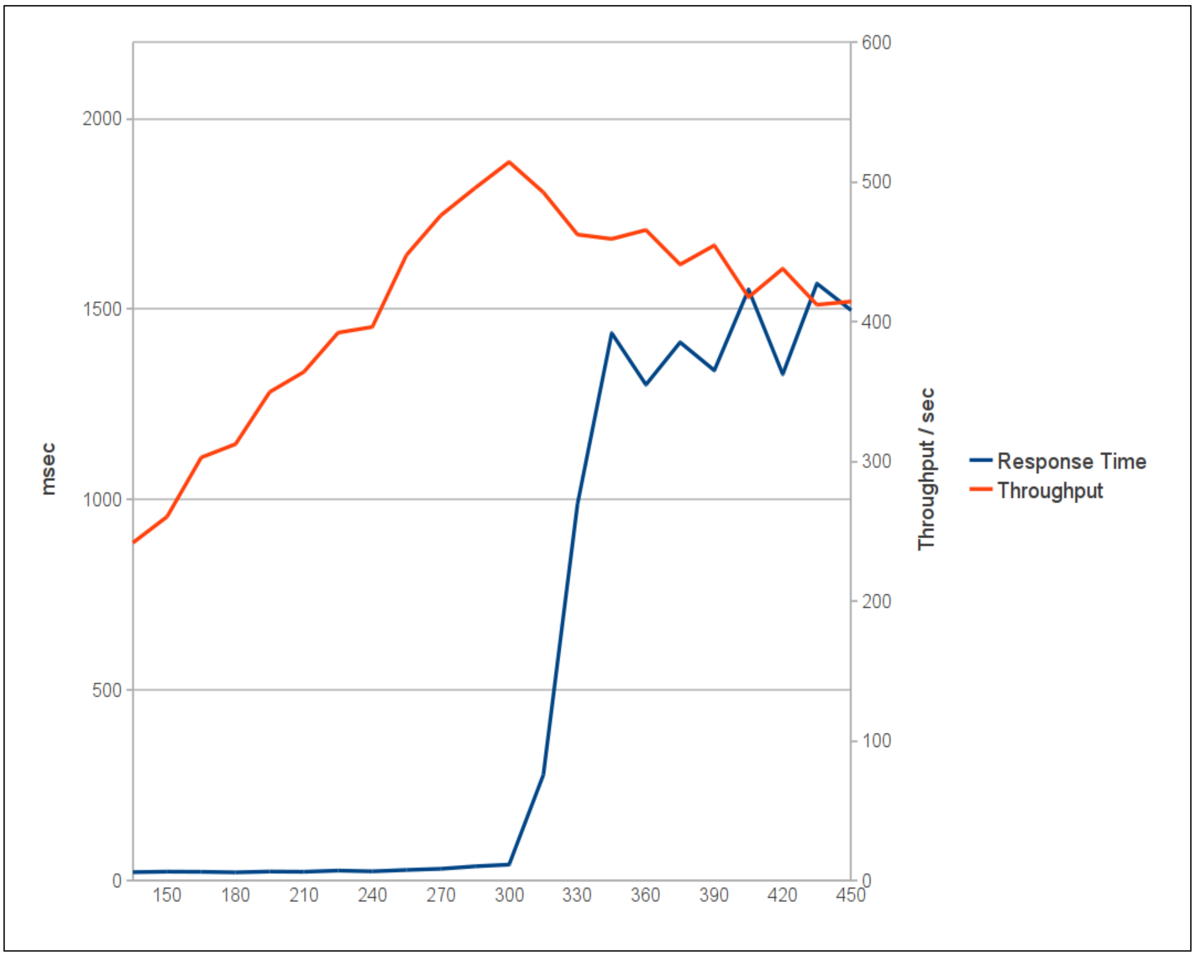

Figure 15: Web Framework, Message Bus and Planet Erlang separated

6 Known Issues

This section describes the issues that we have encountered during our project.

6.1 Erlang Web

We had problems running Erlang Web 2.5.1 initially, getting it to use Yaws 2.4.1 was problematic and it refused to run in non-interactive mode throughout the project. It is hard to tell exactly what is going wrong when we run Erlang Web since the two starting scripts are very different from each other, but

- start.erl is not named properly.
- The included distribution of yaws that is bundled with is compiled for a different architecture than what we were running.

We have not changed the way Erlang Web handles templating at all, but it turned out that to do so would be quite simple, since the template expansion is kept in different modules that you can call without too much fuss. We have not implemented any way for services to add templates to the web framework. The administrator of the system will have to put them there manually as it is now, but it would not be too hard to add that. The problem with making a very dynamic system is that there are few applications which need these kinds of things.

6.2 Erlang Web Bug

During our stress testing 5.2, when we pushed Erlang Web 2.5.1 hard enough it eventually crashed and started to restart itself. So far so good, but after restarting itself and after our stress test was over, Erlang Web was "hogging" one of the CPU cores on the testing machine and would keep it at 100% for roughly 30 minutes until it let go(or you forcibly restarted it).

We have included the tsung.xml C file we used to stress test in the appendix.

6.3 Ecouch

Ecouch [9] is an API for CouchDB 2.2.2 and the API that we chose to use for our project. It is not without its issues. The trouble with Ecouch is twofold. Those two problems are; it is slow and it lacks necessary functionality.

Firstly, Ecouch is unable to, on its own, remove a field from a database with ease. It is a possibility to request the entire database, pluck out the field and then insert the entire database again sans the field you want to be removed, but the solution is neither efficient nor elegant.

Secondly, the performance issue stems from the fact that Ecouch utilizes the http-API of CouchDB rather than calling CouchDB directly (Another CouchDB API, Hovercraft [10], does this). This means that Ecouch has to create http requests and push them through a socket to CouchDB's listener. This is a slow process.

On top of that, Ecouch is itself split up between several processes and must spend some time communicating with itself with every call you make to it. Both these issues could be averted by switching to Hovercraft and refactoring it to make it work like a proper API and not just a wrapper for CouchDB.

The reason why we did not use Hovercraft is also twofold. Firstly, it is more of a wrapper than it is an API. We would not be able to detach it from the CouchDB process and would have to write a bridge to it. Secondly, we could not get views to work properly with it.

6.3.1 Adding new views

Currently, the only way to add new views to the database is via the CoudhDB 2.2.2 web interface, there is no API functionality for that.

6.4 The CMS

We do not have a good view of what exactly should be on either side of the message queue. The main problem is that we have not had time to analyse the usage of the different parts of the CMS (Content Management System) 3.10, so that we could come up with a good separation of the different parts of it. Presumably, the public part of the CMS has this problem as well. You would probably have to implement an Remote Procedure Call [36] functionality in the Web framework, but we are not sure what function calls need to be routed in this way.

There probably is not too much work to it, but the CMS certainly feels a far way off of working correctly at the moment when you look at it.

6.5 Webcrawler

Due to time constraints, development on the web crawler [37] was cut as it was deemed too time costly for the project.

There are one main issues with the crawler. Namely, the current trapexit.org webcrawler is written in Nitrogen 2.5.2 and is incompatible with our Erlang Web 2.5.1 system. While we laid the groundwork for a Nitrogen to Erlang Web bridge, this side project did not reach a workable state and as such we discontinued work on the crawler.

6.6 Releases

We wanted to make a Release for our final product. We initially figured out with a single release file which involved all our applications. It worked out fine until we emerged with the problem of integrating Couchdb 2.2.2 and RabbitMQ 2.3 to our release package. We then did not focus much on the Release structure. But we arrived a temporary solution, by having individual releases for each application.

6.7 Testing Supervisors and Applications

Testing is always important for any system. It is obvious to check our code to look for bugs and then fix them up. We in our system, have tested all our modules using E-unit 5.1 and we got impressive test coverage reports. We initially had problems for testing our connector modules as it includes more than two components to be tested. We spoofed the components but it didn't worked as we wanted. Meck 2.11 was a useful library written in Erlang came to our rescue. It was very useful in spoofing components. Read more about Meck and E-unit in section 2.11. But all was working good until we started to test our supervisor and application modules. These modules are fairly simple and do not have complex code in them. But this was the problem, like we were unable to figure out how to do unit testing with supervisors and applications.

7 Future work

This entails the future developments about our system.

7.1 Security

We have not looked into making the way that traffic is sent secure, so right now all traffic between different services is unencrypted and open for spying. For our testing this has not really been an issue, but you would definitely want to make sure that the valuable information that you pass around (usernames, passwords etc.) are not open for viewing. A simple approach to do this is to simply have sensitive information that is passed around be encrypted before it is sent in the connector or in the services themselves.

We do not regulate the way that services use the Rabbit message queue, although with secure passwords and usernames it shouldn't be too much of an issue. AMQP (Advanced Message Queuing Protocol) [11] has many ways of making the brokering more secure and less open to abuse, although it is not something that we looked into in our project.

7.2 Distributed Applications

In the system we have built there's no process that ensures that all the different services are running as they should. This may or may not be needed but would of course be a nice thing to have, especially if you intend to distribute the applications over many machines that may have their own share of faults affecting the Erlang 2.1 environment.

A supervision tree was designed and proposed with the aim of it being developed, as can be seen in figure 16. However, not much time was spent on implementing this feature, as we were focusing on making the system work and working on the regression of the system. In the end, it goes unimplemented, but would be interesting to add. You could use the RabbitMQ 2.3 ping function to implement this.

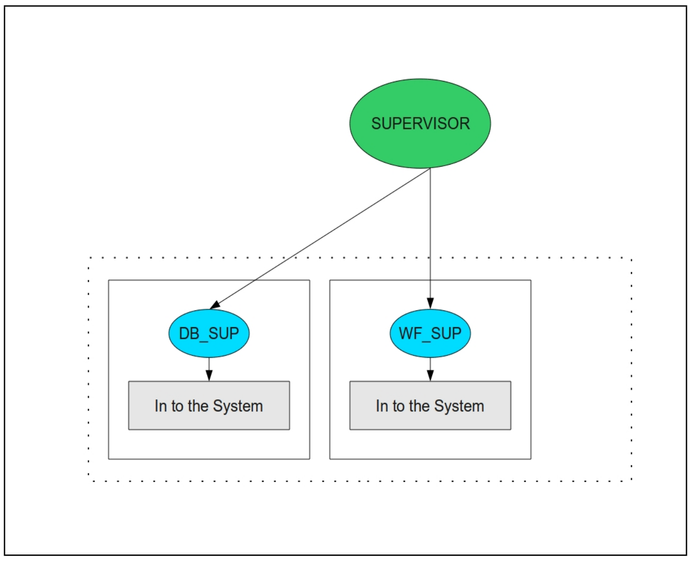

Figure 16: which would enable super-service supervising.

7.3 More services

As far as system is growing new services might be developed and integrated to the system. There are number of services:

- Forum
- Wiki
- Content Management System (CMS)

To add a new service to the system developer has to:

1. implement new callback module;
2. implement new external API;
3. implement new database schema (optional)

7.3.1 Forum Database Schema

In case we were to implement at forum, we worked out a sketch of a database schema for a forum. Even though CouchDB 2.2.2 does not explicitly support relationship between data, you can impose that relationship via clever restrictions and features in the API. This is a brief sketch of how we would do that.

A Database schema for forum service is not as simple as for authorization for example. The data is stored in different databases and there is relations between data as well. Forum usually consists of number of topics. Each topic consists of number of threads and each thread refers to the number of posts. Thus, the architecture is represented by four levels where each level corresponds to separate database (see figure 17).

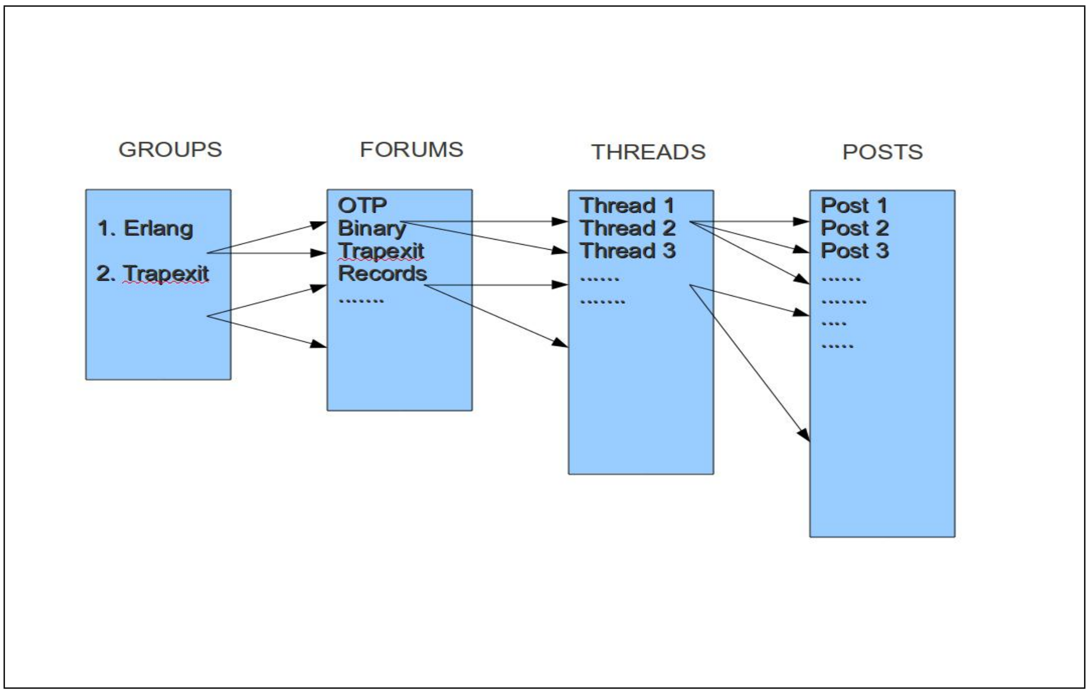

Figure 17: A sketch of the proposed forum schema

Each document in 'groups' database contains references to the documents from 'forums' database. Detailed document structure is:

field_name	value
doc_id	"1098"
group_name	"Erlang"
list_of_references	["1","15", "642"]
db_name	"forums"

where 'list_of_references' is the list of document ids and 'db_name' is the database name where these ids are stored. To get all the forums for the given group a view is used.

If a new service is added to the system then the new database schema should be defined and implemented.

7.4 MQ (Message Queue)

The implementation of our message bus API has been sufficient for our uses, considering for example that we have had no worries about security and no real issues with network latency between our components. To expand the API and make more use of the AMQP (Advanced Message Queuing Protocol) [11] that RabbitMQ 2.3 implements would probably be time well spent, especially when considering the security of the traffic between components. Synchronous calls (I.e. calls that return the response of the recipient) should also, logically, be implemented in the RabbitMQ API, although it is currently implemented in the gen_connector API. Other possible expansion of the RabbitMQ API is the addition of various priority levels that we considered but never implemented and variable timeout settings. The API is minimalistic but functional right now.

7.5 Testing Exchangeability of Components

While we designed our system to have exchangeability of components, complete with a double abstraction 3.8 of all our components we have, mainly due to time constraints, not tested this yet. A reasonable future work would be to swap out the Message Queue or the Database and see how much work is required to make the system run again. In theory, you should only have to rewrite the API's of the module or service you are swapping out.

8 Conclusion and analysis

Throughout this project, the goal has, in a sense, been to explore the "cloud" architecture when applied to a web server. In that regard, we have definitely succeeded. We use an architecture which is by nature distributable to put together components and achieve some not too shabby results in a near-zero latency environment. Considering that these components themselves are also distributable (though we haven't explored and tested the performance effect of doing this) it would definitely be possible to relegate work to many machines.

We've designed and implemented a system that allows for many concurrent actors to work in parallel. The traffic going between is structured to be efficient and lean so that the message bus does not bottleneck the system, and according to our initial testing, that doesn't seem to be the case. In fact, the part that goes down first is Erlang Web 2.5.1, and although it eventually recovers, a bug caused the CPU usage to go up to 100% on one of the cores on the machine running Erlang Web when we did the stress tests as well as afterwards. This is detailed further in the "Known Issues" chapter.

Although we have not tested it yet, we do believe that we have achieved exchangeability of components. Taking out the, say, the Message Queue and putting in another one should only require a rewriting of the mq_api and after that the system should be good to go. I use the word "should" here, because we did actually not try this out ourselves.

References

[1] J.Chris Anderson, Jan Lehnardt and Noah Slater. (2009) *CouchDB: The definitive Guide.* 1st ed. O'Reilly Media. pp.230.

[2] Meck (2011) [Online]. Available from: `https://github.com/eproxus/meck`. [Accessed 25/02/2011].

[3] Francesco Cesarini and Simon Thompson. (2009) *Erlang Programming-A Concurrent Approach to Software Development.* 1st ed. O'Reilly Media.

[4] RabbitMQ (2011) [Online]. Available from: `http://www.rabbitmq.com/`. [Accessed 11/02/2011].

[5] Yaws (2011) [Online]. Available from: `http://yaws.hyber.org/`. [Accessed 11/02/2011].

[6] Erlang Web (2011) [Online]. Available from: `http://www.erlang-web.org/`. [Accessed 11/02/2011].

[7] Redmind (2011) [Online]. Available from: `http://www.redmine.org/`. [Accessed 17/02/2011].

[8] Tsung (2011) [Online]. Available from: `http://tsung.erlang-projects.org/`. [Accessed 18/02/2011].

[9] Ecouch (2011) [Online]. Available from: `http://code.google.com/p/ecouch/`. [Accessed 25/02/2011].

[10] Hovercraft (2011) [Online]. Available from: `https://github.com/jchris/hovercraft`. [Accessed 25/02/2011].

[11] Advanced Message Queuing Protocal (2011) [Online]. Available from: `http://www.amqp.org/confluence/display/AMQP/Advanced+Message+Queuing+Protocol`. [Accessed 25/02/2011].

[12] Nitrogen (2011) [Online]. Available from: `http://nitrogenproject.com/`. [Accessed 25/02/2011].

[13] Zotonic (2011) [Online]. Available from: `http://zotonic.com/`. [Accessed 25/02/2011].

[14] Riak (2011) [Online]. Available from: `http://wiki.basho.com/`. [Accessed 25/02/2011].

[15] Erlang Solutions Ltd. (2011) [Online]. Available from: `http://www.erlang-solutions.com/`. [Accessed 25/02/2011].

[16] Ubuntu (2011) [Online]. Available from: `http://www.ubuntu.com/`. [Accessed 25/02/2011].

[17] Hibari (2011) [Online]. Available from: `http://nosql.mypopescu.com/post/865670585/hibari-cloud-database-a-new-key-value-store`. [Accessed 25/02/2011].

[18] ZeroMQ (2011) [Online]. Available from: `http://www.zeromq.org/`. [Accessed 25/02/2011].

[19] C++ (2011) [Online]. Available from: `http://en.wikipedia.org/wiki/C%2B%2B`. [Accessed 25/02/2011].

[20] Mochiweb (2011) [Online]. Available from: `http://groups.google.com/group/mochiweb`. [Accessed 25/02/2011].

[21] Inets (2011) [Online]. Available from: http://www.erlang.org/doc/man/inets.html. [Accessed 25/02/2011].

[22] HTML (2011) [Online]. Available from: http://en.wikipedia.org/wiki/HTML. [Accessed 25/02/2011].

[23] MVC model (2011) [Online]. Available from: http://en.wikipedia.org/wiki/Model%E2%80%93View%E2%80%93Controller. [Accessed 25/02/2011].

[24] Ajax (2011) [Online]. Available from: http://en.wikipedia.org/wiki/Ajax_(programming). [Accessed 25/02/2011].

[25] Chicago Boss (2011) [Online]. Available from: http://www.chicagoboss.org/. [Accessed 25/02/2011].

[26] PostgresQL (2011) [Online]. Available from: http://www.postgresql.org/. [Accessed 25/02/2011].

[27] RSS (2011) [Online]. Available from: http://en.wikipedia.org/wiki/RSS. [Accessed 25/02/2011].

[28] XML (2011) [Online]. Available from: http://en.wikipedia.org/wiki/XML. [Accessed 25/02/2011].

[29] Ibrowse (2011) [Online]. Available from: https://github.com/cmullaparthi/ibrowse/wiki/ibrowse-api. [Accessed 25/02/2011].

[30] Rebar (2011) [Online]. Available from: https://bitbucket.org/basho/rebar/wiki/Home. [Accessed 25/02/2011].

[31] Git (2011) [Online]. http://git-scm.com/. [Accessed 25/02/2011].

[32] EDoc (2011) [Online]. Available from: http://www.erlang.org/doc/apps/edoc/index.html. [Accessed 25/02/2011].

[33] Uppsala University (2011) [Online]. Available from: http://www.uu.se/. [Accessed 25/02/2011].

[34] Eunit (2011) [Online]. Available from: http://www.erlang.org/doc/man/eunit.html. [Accessed 25/02/2011].

[35] Content Management System (2011) [Online]. Available from: http://en.wikipedia.org/wiki/Web_content_management_system. [Accessed 25/02/2011].

[36] Remote Procedure Call (2011) [Online]. Available from: http://en.wikipedia.org/wiki/Remote_procedure_call. [Accessed 25/02/2011].

[37] Webcrawler (2011) [Online]. Available from: http://en.wikipedia.org/wiki/Web_crawler. [Accessed 25/02/2011].

[38] Open Office (2011) [2011]. Available from: http://www.openoffice.org/. [Accessed 25/02/2011].

A Appendix A: User Manual

A.1 Installation and setup

The four components are required to install:

- CouchDb [1]
- Ecouch [9]
- ErlangWeb [6]
- RabbitMQ [4]

A.1.1 Installation

Before the installation
Be sure that the follow components are installed to your OS:

1. libssl-dev
2. ssh
3. libcurses5-dev
4. git
5. Erlang (R14B or later)
6. mercurial

Installation scripts are placed at trapexit/scripts/install
Put the scripts

1. inst_mother.sh
2. inst_couchdb.sh
3. inst_ecouch.sh
4. inst_erlweb.sh
5. inst_rmq.sh

to your installation folder.

Warnings!
Do not use space in the folder name!
Before running the script DELETE all previous installations!

Start the main script, type: sudo bash inst_mother.sh
Follow the installation instructions.

After the installation
Go to the current installation folder.

1. Change the access permission, type: sudo chown -R user_name:user_name . (do not forget the dot at the end!)
2. Fix Erlang-Web:

(a) run in terminal: export ERL_LIBS='/usr/lib/erlang/lib/inets-5.2'

(b) replace two files:
copy setuid_drv.so and yaws_sendfile_drv.so into /erlang-web/lib/yaws-1.85/priv/lib
or (copy from .../lib/yaws-1.85/priv/lib/* lib/yaws-1.85/priv/lib/)

(c) go to current installation folder/erlang-web

(d) to compile: ./bin/compile.erl
if you get en error "can't find include lib "inets/src/httpd.hrl" then delete files:
   i. lib/eptic-1.4.1/src/e_mod_inets.erl
   ii. lib/eptic_fe-1.0/src/e_fe_mod_inets.erl
   iii. lib/ewgi-0.2/src/ewgi_inets

(e) to setup: ./bin/start.erl

(f) to start interactively: ./bin/start_interactive yaws
if got en error then do again: copy setuid_drv.so and yaws_sendfile_drv.so into /erlang-web/lib/yaws-1.85/priv/lib

3. Restart the computer

A.1.2 Start the system

Before running the system write correct paths into trapexit/scripts/start/path.cfg
Go to the trapexit (root) directory.
Note! All the commands should be run from the 'trapexit' directory!

Start RabbitMQ
Run the RabbitMQ 2.3, type in a shell: bash ./start.sh rabbit
To see RabbitMQ Server interface open web browser and go to http://localhost:55672/mgmt/
username is guest, password is guest

Add users
Go to tab 'Users' and put:

- username: db
- password: db
- administrator: no
- press 'add new user'

Repeat the procedure for the users: 'cms', 'planet', 'rss', 'wf', 'ws', 'auth'

Set permissions
In the RabbitMQ web interface, in the table 'Users'

- click to the user 'db'
- choose 'Set Permission'
- click 'Set Permission' button

Repeat the procedure for all users.

Start CouchDB

To start CouchDB 2.2.2 type in a shell: bash ./start.sh couch
To see CouchDB web interface open web browser and write: http://localhost:5984/_utils
Login as Admin: username 'admin', password 'admin'
To check if CouchDB works properly start 'Test Suite'
If CouchDB does not work run the script trapexit/scripts/couch_repair.sh

To see Erlang-Web web interface open web browser and go to: http://localhost:8080/

Start other components
To start the system without RabbitMQ 2.3 and CouchDB type in a shell: $ bash ./start.sh trapexit

Other modes
To start all the modules: bash ./start.sh all
To start CouchDB only: bash ./start.sh couch
To start Ecouch application only: bash ./start.sh db
To start RabbitMQ module only: bash ./start.sh rabbit
To start Authentication application only: bash ./start.sh auth
To start Web framework only: bash ./start.sh wf
To start Planet Erlang application only: bash ./start.sh planet
To start Content Management System application only: bash ./start.sh cms
To start RSS 2.6 application only: bash ./start.sh rss

A.1.3 DB schema installation

After CouchDB is run the DB schema can be installed.

Step 1:
    Run the Erlang in the terminal in root directory:
    erl -pa lib/*/ebin scripts/install/
Step 2:
    Start the Ecouch application:
    inst_db_schema:start().
Step 3:
    Run the installation instructions:
    inst_db_schema:create().
Repeat steps 2 and 3 for the script: inst_cms_schema.erl

A.2 Writing new services

A.2.1 Implementing a new callback module

The callback module is the main module of a service which handles requests and calls the internal API.

A.2.2 Implementing external and internal APIs

The internal API intends for work with its own service. The external API intends for work with other services. In other words, the external API should be as much general as possible while the internal API should be service specific. By doing this we achieve component replaceability. If we decide to change the current database to other databases we need to change the database internal API only.
If service A wants to communicate with service B, the service A will use service B's external

API. For instance, if the authentication service needs to send a request to the database, it calls the database external API. The internal and external APIs are placed on different sides of the message bus.

A.2.3 Folder structure

The service folder is placed under trapexit/lib folder following the OTP standard.

- /src contains the source code
- /include contains the hrl files
- /test contains the eunit test cases
- /doc contains edoc files
- /ebin contains .beam files
- /release contains boot scripts for release

A.2.4 Compilation with Rebar

Rebar 2.7 has a configuration file called rebar.config. To compile the new service with Rebar:

- adding a new rebar.config file in the service's folder, e.g. /db/rebar.config. This file contains all dependencies.
- registering the new service in file trapexit/rebar.config.
  You need to add the full path to your new service folder here, i.e. "lib/'service_name'/"

A.2.5 Application behaviour

All new services should be implemented as applications. Each application starts its own supervisor and the supervisor starts the general connector. If the new service needs to store its data in the database, new database schema needs to be added.

B APIs

B.1 MQ API

The MQ API is bloated and half done.

function	notes
connect/3	
send/4	
disconnect/1	
subscribe/1	
unsubscribe/1	
flush/1	Not really used, generally it's better to subscribe for efficiency reasons.
ping/2	Not implemented yet.
recv/1	See flush/1

B.2 WF_API

function	notes
load_tpls/1	
set_session/2	
get_session/1	
end_session/1	
validate/3	

B.3 DB_API

function	notes
read/3	
info/1	
view/3	
insert/2	
insert/3	Third parameter is with field_list
write/3	
size/1	

B.4 AUTH_API

function	notes
check_passsword	
encrypt_password/1	
insert_user/2	
change_password/3	

B.5 gen_connector

function	notes
start/1	
stop/1	
send/3	
call/3	

B.6 db_api

function	notes
read_obj/3	
read_group/3	You have to insert these manually. See chapter 6.3.1.
insert/3	
delete/2	
update/3	
size/1	

B.7 auth_api

function	notes
check_password/3	
encrypt_password/1	This function is not (as currently implemented) deterministic
authenticate/2	
change_password/3	
insert_user/2	

B.8 e_auth and e_db

These follow the definitions as found in Erlang Web (to the best of our knowledge).

C Tsung.xml for stresstesting

This is the stress test that produced the CPU core lockdown issue detailed in the Erlang Web Bug section in our Known Issues chapter.

```
<?xml version="1.0"?>
<!DOCTYPE tsung SYSTEM "/usr/local/share/tsung/tsung-1.0.dtd">
<tsung loglevel="notice" version="1.0">

  <!-- Client side setup -->
  <clients>
    <client host="localhost" use_controller_vm="true"
            weight="5" maxusers="10000"/>
  </clients>

  <!-- Server side setup -->
  <servers>
    <server host="130.238.15.220" port="8080" type="tcp"></server>
  </servers>

  <!-- to start os monitoring (cpu, network, memory). Use an erlang
       agent on the remote machine or SNMP. erlang is the default -->
  <monitoring>
    <monitor host="myserver" type="snmp"></monitor>
  </monitoring>


  <load>

    <arrivalphase phase="1" duration="20" unit="second">
      <users arrivalrate="150" unit="second"/>
    </arrivalphase>

    <arrivalphase phase="2" duration="20" unit="second">
      <users arrivalrate="180" unit="second"/>
    </arrivalphase>

    <arrivalphase phase="3" duration="20" unit="second">
      <users arrivalrate="210" unit="second"/>
    </arrivalphase>

    <arrivalphase phase="4" duration="20" unit="second">
      <users arrivalrate="240" unit="second"/>
    </arrivalphase>

    <arrivalphase phase="5" duration="20" unit="second">
      <users arrivalrate="270" unit="second"/>
    </arrivalphase>

    <arrivalphase phase="6" duration="20" unit="second">
      <users arrivalrate="300" unit="second"/>
    </arrivalphase>

    <arrivalphase phase="7" duration="20" unit="second">
      <users arrivalrate="330" unit="second"/>
    </arrivalphase>

    <arrivalphase phase="8" duration="20" unit="second">
      <users arrivalrate="360" unit="second"/>
    </arrivalphase>

    <arrivalphase phase="9" duration="20" unit="second">
      <users arrivalrate="390" unit="second"/>
    </arrivalphase>

    <arrivalphase phase="10" duration="20" unit="second">
      <users arrivalrate="420" unit="second"/>
    </arrivalphase>

    <arrivalphase phase="11" duration="20" unit="second">
      <users arrivalrate="450" unit="second"/>
    </arrivalphase>

  </load>

  <options>
    <option type="ts_http" name="user_agent">
      <user_agent probability="80">
        Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.7.8)
        Gecko/20050513 Galeon/1.3.21</user_agent>
      <user_agent probability="20">
        Mozilla/5.0 (Windows; U; Windows NT 5.2; fr-FR;
        rv:1.7.8) Gecko/20050511 Firefox/1.0.4</user_agent>
    </option>
  </options>

  <sessions>
    <session name='dumb-login'
             probability='100'
             type='ts_http'>
      <request><http url='http://130.238.15.220:8080/login/'
                     version='1.1'
                     method='GET'>
      </http></request>
      <request><http url='/stylesheet.css'
                     version='1.1'
                     if_modified_since='Mon, 22 Nov 2010 10:28:23 GMT'
```

```
                     method='GET'>
      </http></request>
      <request><http url='http://130.238.15.220:8080/script.js'
                     version='1.1'
                     if_modified_since='Mon, 22 Nov 2010 10:28:23 GMT'
                     method='GET'>
      </http></request>
      <request><http url='/www.gif'
                     version='1.1'
                     if_modified_since='Mon, 22 Nov 2010 10:28:23 GMT'
                     method='GET'>
      </http></request>
      <request><http
                  url='/do_login'
                  version='1.1'
                  contents='username=hej&amp;password=hej'
                  content_type='application/x-www-form-urlencoded'
                  method='POST'>
      </http></request>

      <thinktime value="1" random="false"/>

      <request><http
                  url='/logout'
                  version='1.1'
                  method='POST'>
      </http></request>

    </session>
  </sessions>
</tsung>
```