Twitter
An Architectural Review

Matthijs Neppelenbroek
0331716
M.G.Neppelenbroek@students.uu.nl

Matthias Lossek
F100132
Matthias.Lossek@student.uni-augsburg.de

Rik Janssen
3549402
R.B.M.Janssen1@students.uu.nl

Tim de Boer
0237884
T.deBoer@students.uu.nl

Software Architecture
Faculty of Science
University of Utrecht

January 14th 2011

Contents

1 Introduction

Social networks and community websites became more and more important over the last years. Big players on the market like Facebook[1] and Twitter[2] got so many active members, that these platforms even have an effect on daily life of modern societies [8]. The youth of today tweets, pokes, follows and posts on walls instead of writing emails, letters or calling their friends. Tons of private messages, knowledge and feelings get published every second in the data universe.

These interactions between millions of people are a huge technical challenge for the companies. The users wouldn't use these services if they can't rely on their real time performance. Though in an environment with a smaller amount of users this real time aspect is simple to handle, it becomes a hardly satisfiable requirement while the amount of users grows keenly.

But how can that efficiency in such big software projects be assured and which trade-offs have to be solved? This paper gives an internal view about the software architecture of Twitter and the trade-offs Twitter had to deal with in the past years.

Twitter is an interesting product for analysing its architectural design. It is not only one of the largest community websites and number 10 in traffic worldwide [6]. It is based on open source software and frameworks and contributes its internal projects to the open source community which gives us the opportunity to understand and reconstruct the exact architecture.

But still Twitter itself is not open source! There are only spare official information available how the open source components work together and in what way they are adapted for Twitter's needs. For that reason most of the information about Twitter's internals comes from Twitter's developers' blogs and other community discussions on the Internet, where Twitter's main developers deliver an insight into why and how architectural decisions has been made during the evolution of Twitter.

Next the function of Twitter is described in more detail, then we explain which software quality aspects are important and how they evolved.

2 What is the Function of Twitter?

Before we to define which particular software quality aspects are important for Twitter we need to know for what kind of functionality it is designed.

[1] `http://www.facebook.com/`

[2] `http://www.twitter.com/`

The idea of Twitter came from Jack Dorsey in 2006 who saw that people wanted to share their current activities with others. Before Twitter people shared this information via instant messaging programs or via texting. Dorsey combined the functionality of communicating from one to many persons, that was enabled by the instant messaging, with the functionality of SMS texting. This enabled users of the new system to say what they are doing, anytime and 'anywhere' in the world. Twitter was born.

The official term for telling what you are doing is called microblogging, the length of the messages is constraint by the maximum number of characters one SMS text-message may contain. Microblogging is the main functionality but there are other important features that are important for Twitter. The second important functionality is that of following other users being getting updates about the messages, called tweets, that users that you are following have posted. This enables a user to get easily information about friends, family and other interesting people.

There has been a slight change in focus on what kind of information the tweets on twitter should contain. Before November 2009 the question that was answered by the users of Twitter was "What are you doing?" after that it became "What's happening?". This change of focus ensures that more and more information on events is posted on Twitter instead of information about persons. This makes the change bigger that valuable information about events can be found on Twitter and for making finding this information easier the third the three most important functionalities is the search-function. By using hash-tags in Tweets people can give a subject to a particular message and via this subjects tweets can be and more information comes available to users of Twitter but also anyone else on the web.

3 Architectural Description

To describe the architecture, we make use of Kruchten's "4+1" view model [9]. The 4+1 View Model organizes a description of a software architecture using five concurrent views, each of which addresses a specific set of concerns, logical, process, physical and development view. The fifth view, scenario view, is used to illustrate and validate the other views.

We describe Twitter's architecture by taking Kruchten's view model into account. The next subsections will encounter each view of the model individually.

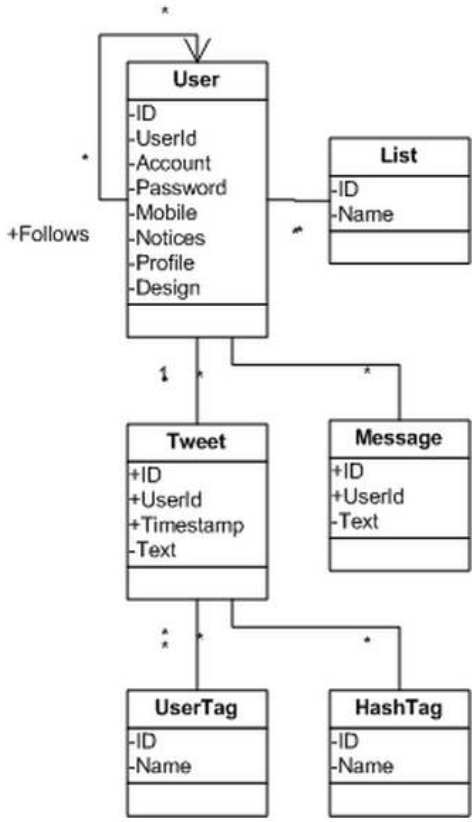

Figure 1: Twitter class diagram

3.1 Logical View

The logical view explains the functionality that the system provides to end-users. Although this is partly covered in the previous chapter, this section adds a class diagram to describe the structure in terms of a class diagram (see figure 1).

The primary class of Twitter's data structure are its Users.

- Users can create Tweets
- Users can send private Messages to other Users
- Users can be grouped using Lists

Tweets are connected to exactly one User. An interesting aspect of Twitter is the followers/following structure, basically Users which are connected to other Users. We are not sure about the exact configuration but it's most likely that the User class can refer to itself.

Hereby Followers (actually Users) are connected to Users.

3.2 Process View

The process view explains the system processes in terms of communication and addresses the behavior of a system in runtime. This is covered in section 10 Scenario Overlay.

3.3 Physical View

The physical view describes the topology of Twitter's software components and their communication. Unfortunately, there is hardly any accurate information on this issue. We could make up a valid UML-diagram but there is too little information about the topology of Twitter's components to make it viable.

3.4 Development View

The development view illustrates Twitter from a perspective of a developer or programmer and is concerned with software management. As Twitter's development view changed significantly over the years this is elaborated in chapter 5. Evolution and quality aspects of Twitter.

3.5 Scenario View

This view illustrates Twitter's architecture by a small set of use cases and scenarios. Interactions and connections between Followers/Following/Tweets and their underlying processes are explained here.

The following use case diagram also contains mandatory functionalities such as logging in to Twitter. The use case is described in table 1

The non-functional requirements for the user perspective are availability, performance and quality. First the site needs to be available before a user can post a tweet, second the user does not want to wait endless before getting a conformation that the tweets is successfully posted. Last but not least the message that the user has entered needs be shown in a way the user had suspected it. If the message misses a particular part or letters are not in the correct order, then the quality aspect of Twitter has severe issues.

To satisfy these non-functional requirements which are the most important for the users of Twitter a trade-off needs to be made with other quality aspects. These aspects are may be not important for the actual user of the system, but can be very important for the other stakeholders. For example

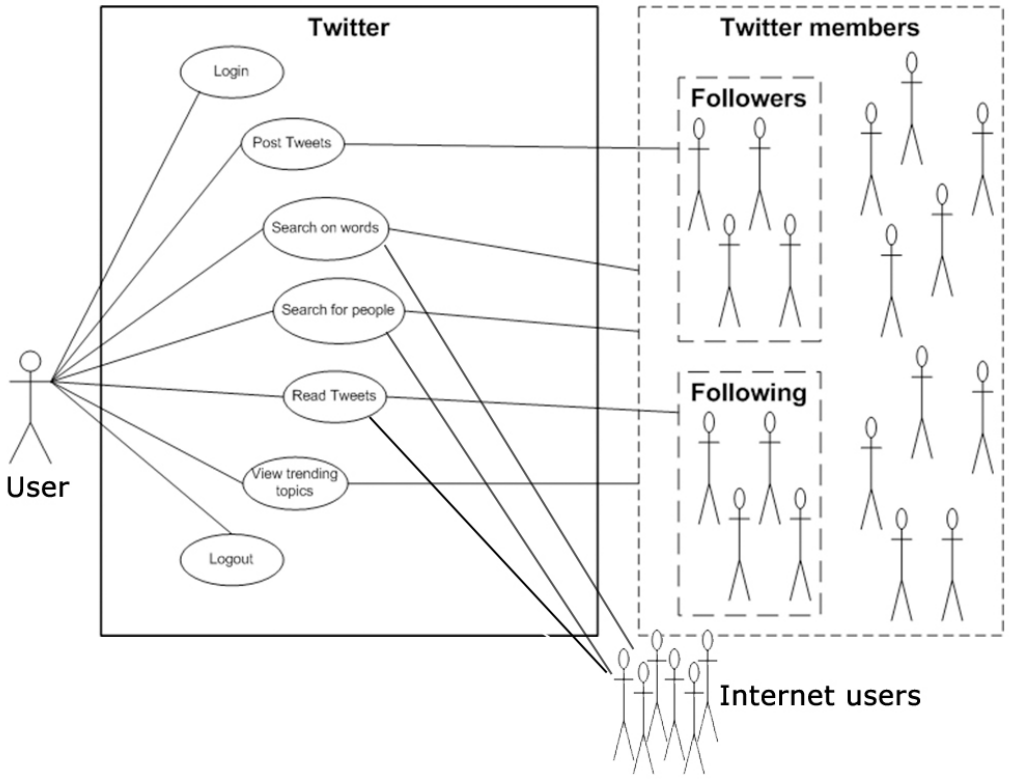

Figure 2: Use case diagram of Twitter.

Use case	**Post Tweets**
Description	Posting a tweet on the account that is own by the person who is posting the tweet.
Actors	User, Followers, the system (Twitter.com)
Pre-conditions	1. User is logged in.
Post-conditions	1.Tweets must be shown on the profile of the user and all his followers. 2.The tweets must contain exactly the same text as entered by the user.
Steps	1. Go to Twitter.com 2. Enter text in to the 'what's happening?' textbox 3. Press the tweet button.
Variations	3.1 A picture is shown of the so called 'Fail Whale' which indicates that there are to many tweets posted at that moment in time. The users needs to try it again.
Non-functional requirements	Availability, performance, quality

Table 1: Use case: Post Tweets

the programmers who need to maintain or update the twitter system our the managers of Twitter.com that need to pay the cost of running thousands of servers to keep Twitter.com going. Which quality aspects are the most important for all the stakeholders and are mostly interacting with each other will be described in the next chapter, quality aspects

4 Quality Aspects

Before we have a look at the quality aspects that are important to Twitter we describe the ISO 9126 standard. This standard provides guidance for making architectural decisions by using six categories of software quality.

4.1 The ISO 9126 Standard for Software Quality

For making architectural decisions during the software design process it is very useful to have categories for rating the software's quality. Some facts have to be specified, if an architecture fulfils the given requirements or if it doesn't. It is not sufficient to say, the software should perform fast or should be reliable. With these terms one could always run into the lack of

precision of natural language. What means "fast"? What means "reliable"? Every person has a different idea about these informal expressions. For example while specifying requirements for a software project, the product owner should qualify in exact numbers, how fast the software has to react for being accepted as a success.

For this reason, the International Organization for Standardization (ISO) released the ISO/IEC 9126 standard for software quality. This standard contains of six categories, each of them divided into subcategories which can be used for describing all possible quality aspects of a software project. The following table shows an overview of all the categories and subcategories specified in the ISO 9126 standard.

The idea is to give each subcategory attributes which can be rated with numbers. These attributes are not given by the standard as they should be specified for each project individually. With this approach, it is possible to compare different solutions in every category. Deciding for a solution is now only finding the optimum of the subcategories' scores. For further reading about the optimisation, we recommend [5], for the ISO 9126 standard in general contact the official ISO document [4].

4.2 Quality Aspects Addressed by Twitter

The importance of software quality aspects differ for each piece of software. The quality aspects that are important to Twitter are reliability, efficiency and maintainability. Over the course of the years, Twitter optimised their quality aspects reliability and efficiency, in several ways. However, the quality aspect maintainability became worse because of those changes. To explain why and how Twitter made certain trade-offs in software quality aspects, we need to have a look at trade-offs in software quality aspects in general.

4.3 Trade-offs in Software Quality Aspects

A trade-off is a situation in which you decrease one quality aspect in return for increasing another quality aspect. Ideally, software design decisions are made with full understanding of the positive and negative implications on its quality aspects.

An example of a trade-off is a highly structured and modularised code that is easy to read by humans, hence easy to maintain. However, because it is highly structured it does not perform as well as less structured code. In this case, a trade-off between Maintainability and Efficiency is made. An increase in Maintainability will decrease the Efficiency of a system and vice verse. This situation can be illustrated by the sliders in figure 3.

Category	**Question behind**	**Subcategories**
Functionality	Are the required functions available in the software?	Suitability Accuracy Interoperability Security Functionality Compliance
Reliability	How reliable is the software?	Maturity Fault Tolerance Recoverability Reliability Compliance
Usability	Is the software easy to use?	Understandability Learnability Operability Attractiveness Usability Compliance
Efficiency	How efficient is the software?	Time Behaviour Resource Utilisation Efficiency Compliance
Maintainability	How easy is to modify the software?	Analyzability Changeability Stability Testability Maintainability Compliance
Portability	How easy is to transfer the software to another environment?	Analyzability Changeability Stability Testability Maintainability Compliance

Table 2: Categories and subcategories of the ISO 9126 standard.

Figure 3: The trade-off problem illustrated with sliders

The first slider shows a trade-off in favour of Efficiency, as the slider is set closer to Efficiency than to Maintainability. The second slider shows a trade-off in favour of Maintainability, as the slider is set close to Maintainability than to Efficiency.

However the second slider indicates that a lot of Efficiency is traded off against Maintainability. The first indicates a more neutral trade-off between Efficiency and Maintainability as the slider is positioned relatively close to the centre of the bar.

4.4 Trade-off 1: Efficiency Versus Maintainability

To improve the efficiency, Twitter increased the variety of technologies being used and they adopted a more complex programming model. These technical changes made the maintenance of Twitter more difficult, thereby trading of efficiency for maintainability. How the trade-off efficiency versus maintainability evolved and the reasoning behind it is described chapter 5 and 6.

4.5 Trade-off 2: Reliability Versus Maintainability

To improve the reliability, the complexity of Twitter's server-infrastructure increased. They started to use more servers and they all had to be balanced and synchronised. Balancing and synchronizing servers using MySQL databases is not the easiest task, thereby reliability for maintainability. How the trade-off reliability versus maintainability evolved and the reasoning behind it is described in chapter 5 and 6.

5 Evolution and Quality Aspects of Twitter

In this chapter we talk about the changes of Twitter's architecture and corresponding quality aspects through out the years. Twitter started out in 2006 as a side project of employees from the Odeo company who build a website for distributing pod-casts.

The people on the project wanted to test the functionality that they had in mind for Twitter as soon as possible. To do this they used Ruby on Rails. The choice for Ruby on Rails was only because the Odeo companies product, websites to post podcasts on, was also build on Ruby on Rails. The software engineers had experience with this programming language and therefore this was the tool for them to make a prototype of Twitter as fast as possible. Ruby on Rails was used to make the website and the interactions

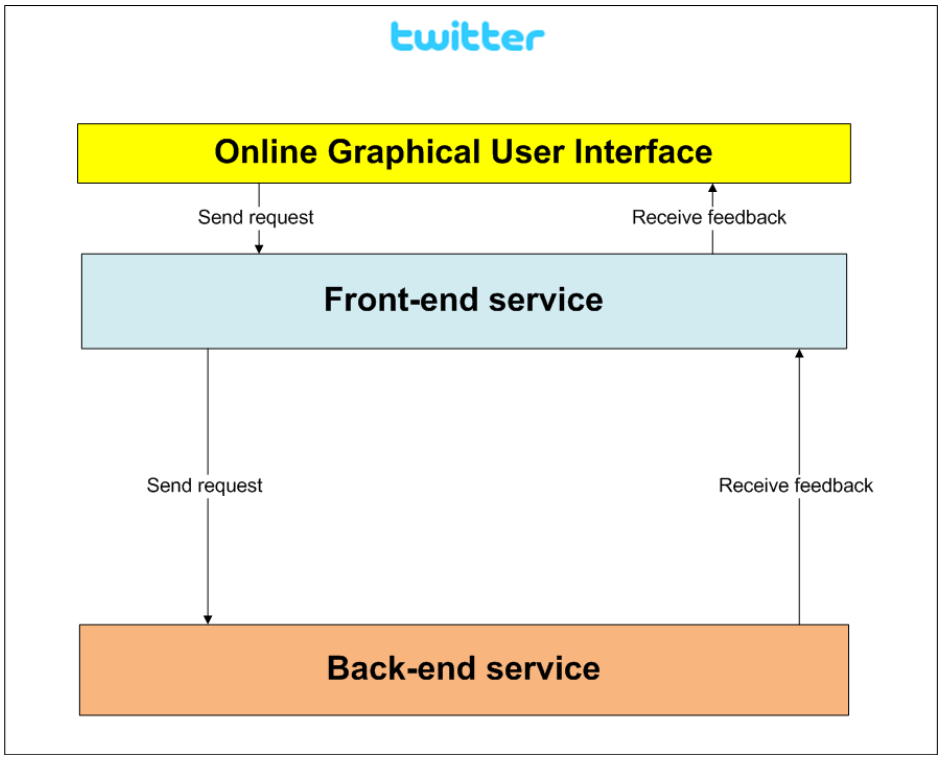

Figure 4: Twitter's architecture at the beginning (2006)

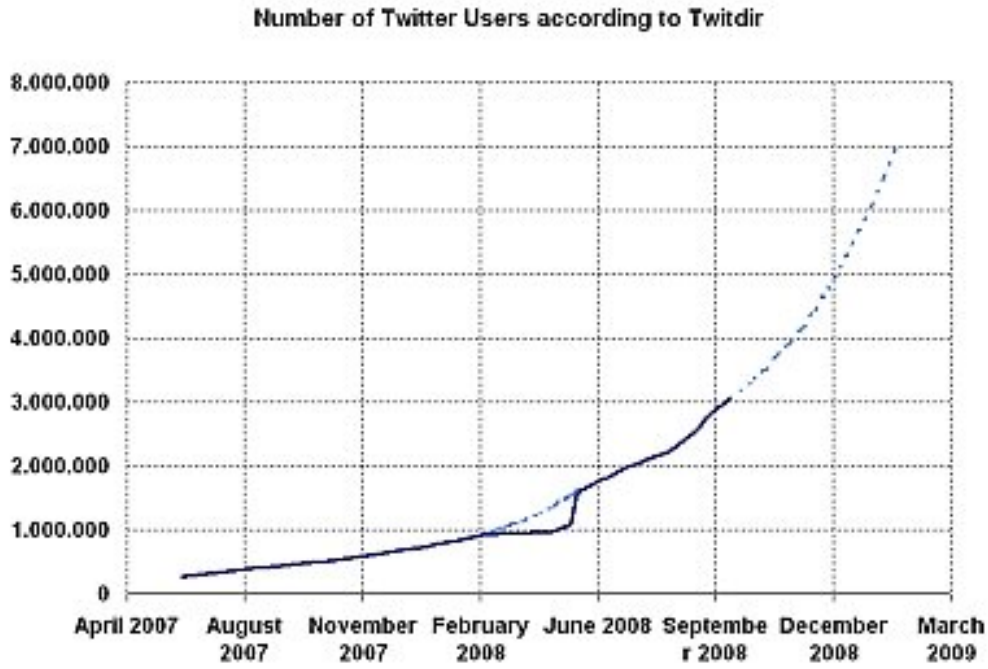

Figure 5: Rapid growth of Twitter.

on it possible. In figure 4 this is called the online graphical user interface. Also the front-end service was written in Ruby on Rails and it provided the communication between the back-end service.

At the beginning of Twitter the back-end service was only a MySQL database that was used to store all the messages that where being posted. With this architecture they could satisfy the users most important software quality aspects, availability, performance and quality. The software engineers building the system could test if the functionality of the website was something people would going to use. At that time nobody had foreseen the immense growth of twitter the following two years, so further improvement of the system was not a priority.

In 2007, Twitter really started to grow more rapidly and this exponential growth continued in 2008 (see figure 5). Twitter is not limited to a website only on which people can post messages, most of the message are posted via mobile devices. This feature is enabled by the API Twitter has written to get easily information in and out of the Twitter system. Via this API feature people can make third-party software which enables users to stay connected to Twitter via SMS. The trade-off for this much used feature is that the architecture of Twitter needed to be improved. The front-end service alone

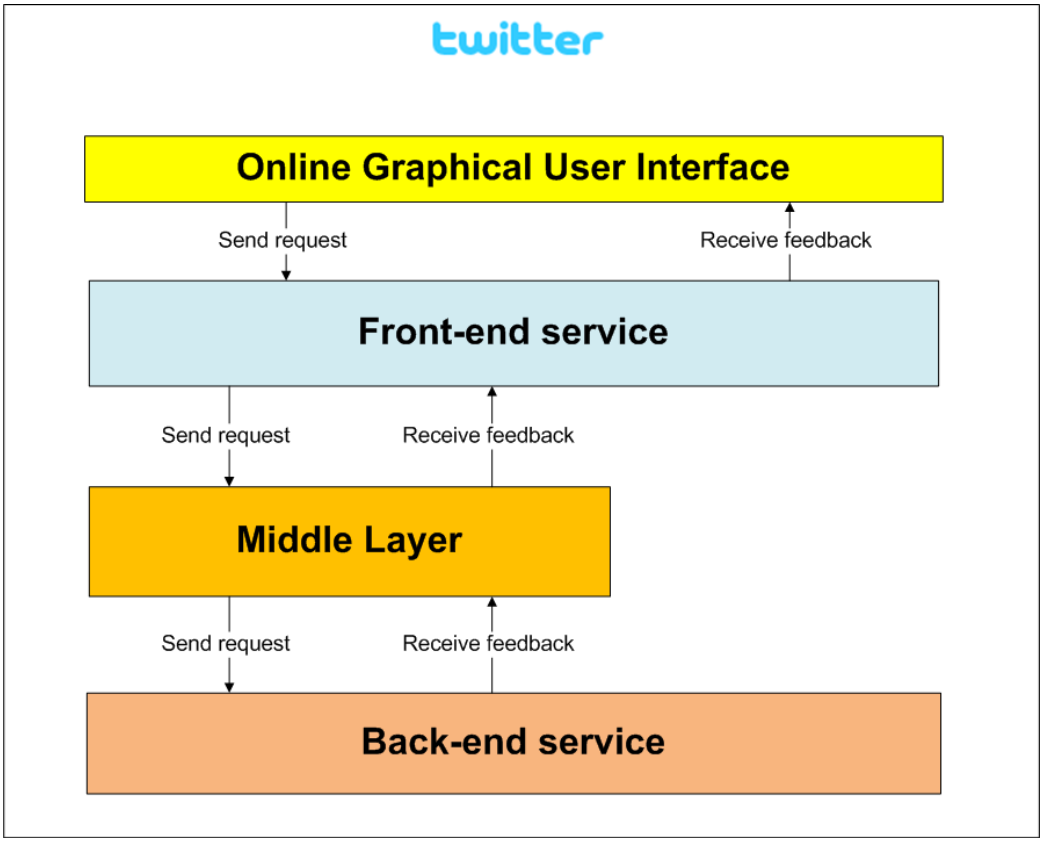

Figure 6: Twitter's architecture (2008) added a middle layer (queuing system)

could not handle all the requests from the website completed with and more important request information from and to a diversity of API's.

The solution was a middle layer (figure 6) that could handle the enormous amount of Tweets posted per second. This middle layer was also written in Ruby on Rails and provided a queuing system in which the tweets could be saved before they where written to the back-end.

6 Twitter's Architecture

In the few years that twitter exists (four years, since 2006), they used many architectures to fit best to their needs. As we saw in the previous section, they introduced a layer structure to handle different quality aspects that are important for Twitter. The architecture depicted in figure 7 is the current architecture they use taking all the current quality aspects into account. A

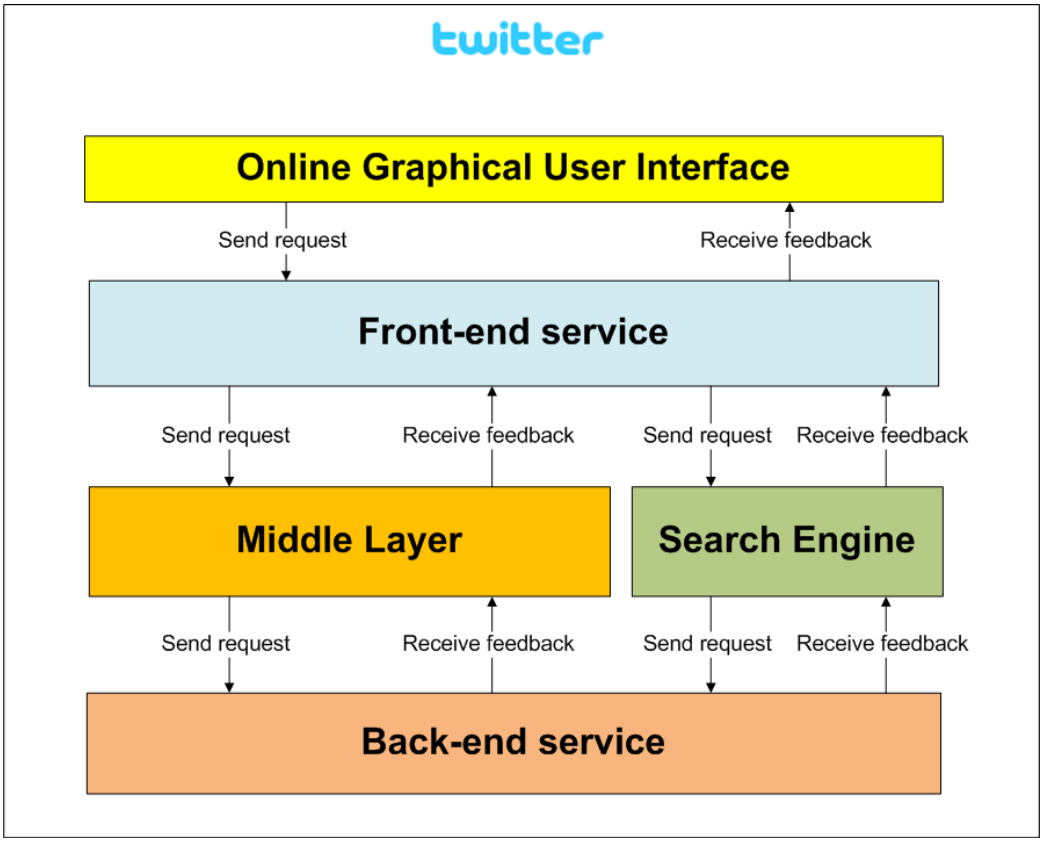

Figure 7: Twitter's current architecture

new search engine is recently implemented to speed up searches dramatically (i.e. improves the efficiency quality aspect), and using the middle layer as a queueing system makes the system more robust (i.e. improving reliability quality aspect).

All layers in Twitter's architecture are implemented using open source software, which is shown in figure 8. In the next subsections we will explain what the purpose of the different layers are by taking the open source software into account that is used to implement those layers.

6.1 Back-end Service Layer

The back-end service layer of Twitter stores all the tweets that are posted by the members. Basically the back-end service layer is only used for inserting and retrieving tweets. As is explained in the previous section they used MySQL for storing the all the data, but this is recently replaced buy Memcached. Memcached is an open source, high-performance, distributed

memory object caching system[3]. Using Memcached, Twitter is able to store all their tweets in-memory, making data retrievals very fast. The MySQL databases that were used earlier, are still in use, but function only as backup system.

6.2 Search Engine Layer

Recently Twitter launched their new search engine. This search engine is implemented using Apache's Lucene[4]. Lucene is a high-performance, full featured text search engine library written entirely in Java. Lucene uses inverted index to index the stored tweets, therefore word searches has a very quick response.

The basic idea of the inverted indexing method, is that text-based sentences (i.e. Tweets) are stored by splitting them into words (i.e. indexing the words of a sentence). The words are stored accompanied by a reference to the corresponding tweet. When performing a word-search, Lucene searches for the word, request the back-end service layer for the corresponding Tweets that store that word (i.e. by using the reference to the tweets) and returns the received tweets.

6.3 Middle Layer

The middle layer is basically used as a queueing system to not overload the back-end service layer, as was the case before when they used no middle layer and only MySQL for storage. The middle layer was first implemented by Starling which is programmed in Ruby on Rails. But as this turned out to be too slow and with a unsatisfactory crash recovery.

The software engineers at Twitter needed to replace the Starling component by a better system. There where a lot of systems that could replace the Starling component. They looked at three other message queues and the option of rewriting the starling component from Ruby on Rails to JRuby, the Java implementation of the Ruby language. JRuby had not much advantages because a lot of rewriting would be needed to get it working and the efficiency of the language was quit poor. JRuby could not compile direct to Java Virtual Machine bytecode but first needed to interpret JRuby to Ruby and then eventually to JVM bytecode. This had a bad influence on the efficiency of the system.

The other three existing queuing system that where examined where ActiveMQ, RabbitMQ and MemcacheQ. The activeMQ was after benchmarking

[3] `http://memcached.org/`

[4] `http://lucene.apache.org`

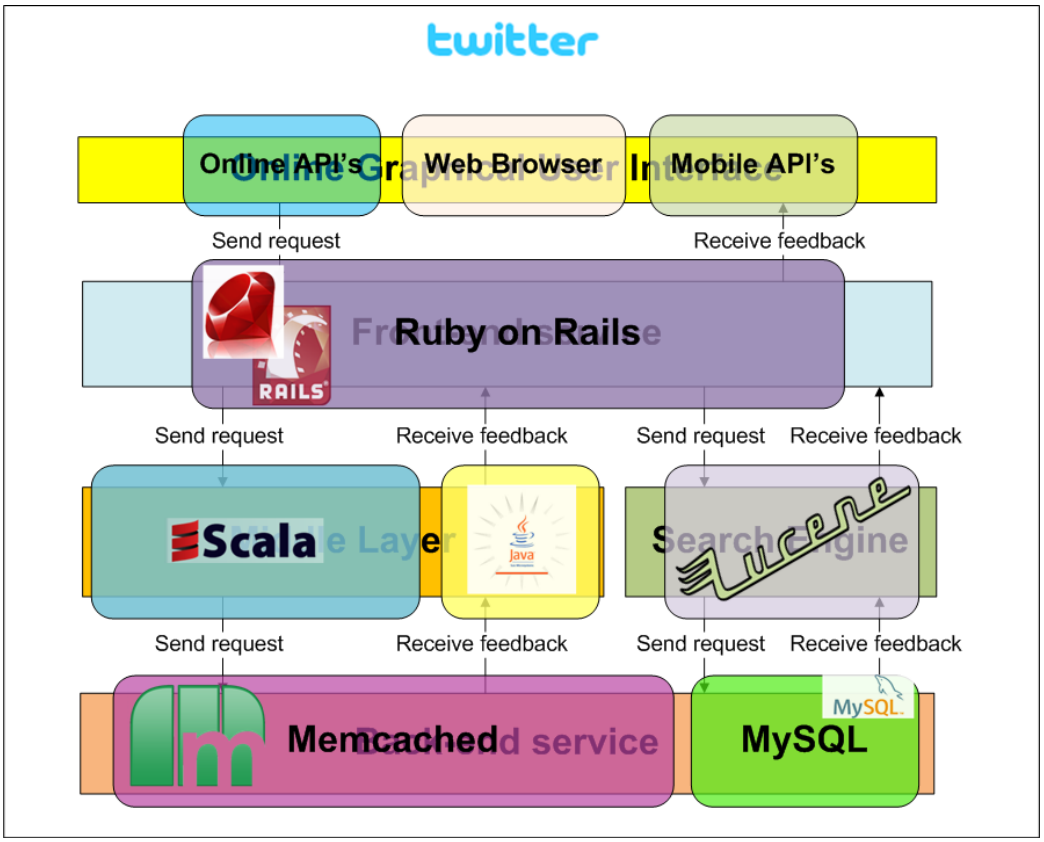

Figure 8: Software overlay

it to slow to use for Twitter. People within Twitter thought RabbitMQ would be the message queue that would solve their problems but unfortunatly this system turn-out to crash when more message where put in the to the queue then there was memory. After discussing this problems with the creators of RabbitMQ they decided to look for an other system.

The solution was found by a message queue system that was build by one of the software engineers from Twitter, named Kestrel. Kestrel is a recode of Starling using Scala, which is much more efficient than the Starling implementation. Another advantage was that also easily to test because they could replace one of the running Starling servers with a Kestrel version and observe how it managed the load it had to deal with. This made it more it possible to gruadually shift from a middle layer made out of Starlings servers to a layer of Kestrel servers. Also Kestrel makes the use of Memcached possible. Because Scala is a general purpose programming language built on JAVA, we also mention JAVA as key software for the middle layer in the software overlay shown in figure 7.

6.4 Front-end Service Layer

The front-end service layer is the front desk for handling all requests to the Twitter system. All requests that are handled by the Twitter system comes through the front-end service layer, which will pre-calculate and delegate the requests for the correct handling through the whole system. The front-end service is build using the Ruby on Rails web framework. Ruby on Rails is a framework based on JAVA which optimizes programming productivity and keeps your programming code sustainable[5].

6.5 Online GUI's

For members to use Twitter, they make use of a wide variety of online graphical user interfaces (GUI's). The main GUI is the website of Twitter where you can logon to the system and use all the Twitter features. But the real power of Twitter is the possibility to also connect to Twitter using an Application Programming Interface or API. Other companies can develop and invest in those API's, which can be used for instance on mobile devices like a Smartphone. The API can make request for data retrieval or do an data insertion request, which is requested to the front-end layer.

[5] `http://rubyonrails.org/`

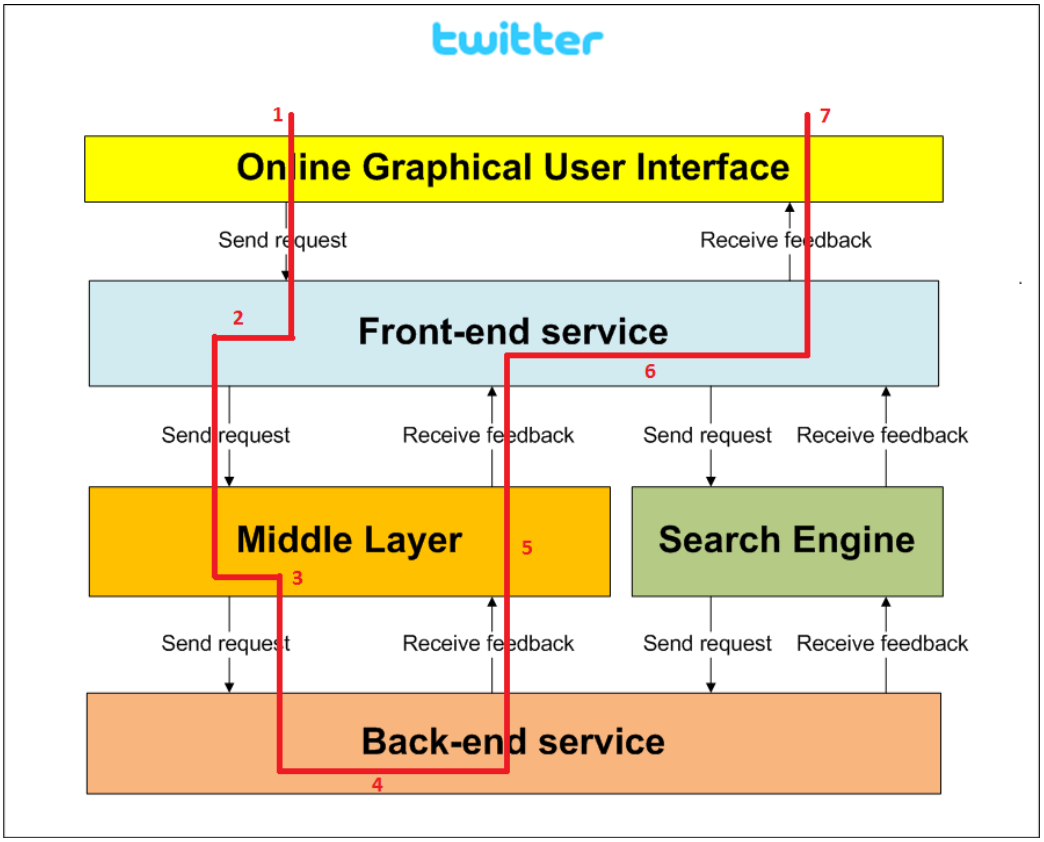

Figure 9: Software architecture scenario overlay (1)

6.6 Scenario Overlay

We will explain how Twitter works by using two scenarios. The scenarios are visualized using a scenario overlay on the software architecture of Twitter introduced in figure 4. The first scenario we will focus on, is the insertion of a tweet (e.g. updating your current status to "Watching TV in bed"), which is visualized in figure 9.

We start by logon to the website or to your favorite API, typing your current status (max 140 characters) and send the tweet (1). The request for the insertion is processed by the front-end service layer (2), which sends the information to the middle layer. The middle layer put the insertion request into the Kestrel queuing system for processing by the back-end service layer (3). When the request in in front of the queue, the tweet is inserted into the Memcached memory in the back-end service layers (4). Now also the search engine is updated by indexing the new tweet into the inverted index memory of Lucene. When the tweet is successfully inserted into the memory

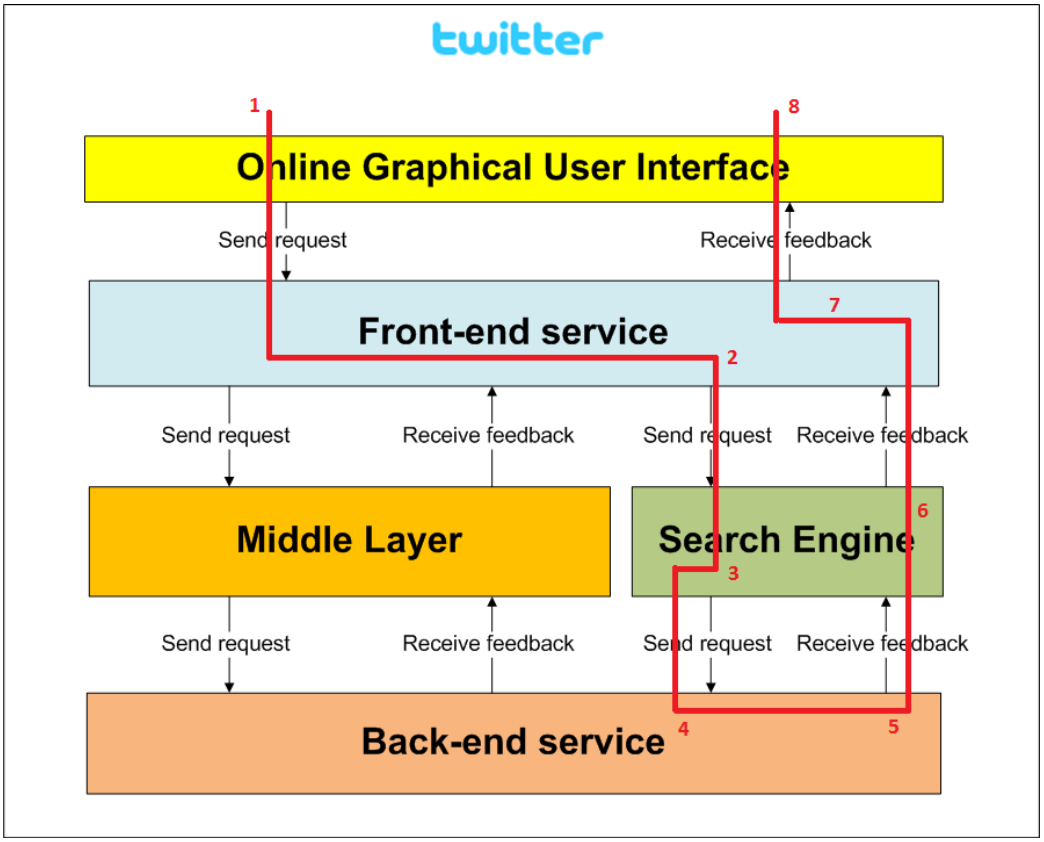

Figure 10: Software architecture scenario overlay (2)

of Memcached, Kestrel, in the middle layer, gets feedback from the back-end service layer that the tweet is stored (5). Kestrel notifies the front-end service layer that the tweet has been successfully inserted (6), which directly sends a message to the GUI that requested the insertion of the tweet.

The second scenario to explain the working of the Twitter system is visualized in the scenario overlay in figure 10, which represent the scenario of a search request (e.g. the search for the word "bed"). The same information apply for the first step (1), but in the front-end service layer the search request is identified and therefore the request is delegated to the search engine (2). The Lucene search engine processes the request which result in a list of corresponding stored tweets which contains the requested word(s) (3). Memcached is requested for getting the tweets from the list (4) which is send back to the search engine (5). The search engine returns the tweets containing the word(s) (6). The front-end service layer get's the tweets which notifies the GUI by sending back the found tweets (7). The GUI will show the tweets containing the word(s) that was searched.

7 Comparison to Similar Systems

As we could see in the previous chapters, Twitter designed its architecture around its needs. They even invented new concepts like their Scala message queueing system Kestrel for solving the scalability problems. There was no suitable solution available even if Twitter is not the only one who had to answer these questions.

From a technical point of view, it is now very interesting to compare Twitter's solution with Twitter's competitors on the microblogging and social network market to see, in which way their architectures differ from Twitter's and why they chose another way for handling the emerging challenges. For this paper we picked two alternatives. First identi.ca[6], a microblogging website with very similar features as Twitter and second Facebook, the global market leader for social networks and online communities.

In the following chapter we will show the similarities and differences between these systems and Twitter.

7.1 Identi.ca

The microblogging website identi.ca has been launched in 2008. It has basically the same features as Twitter. It is possible to write 140 character long status messages on his profile, one can subscribe to the status messages of other identi.ca members and it is possible to tag topics or people, as in Twitter. Identi.ca is owned by StatusNet which is also the name of the open source software project it is based on. So identi.ca is a simple installation of the StatusNet software package. StatusNet is open source and released under the Creative Commons Attribution 3.0 license. It is written in PHP and needs a MySQL database for its back-end.

Figure 11 shows the identi.ca website.

The basic idea behind StatusNet is to provide an alternative system to Twitter, which can be adopted very easily for closed non-public blogging groups. Everyone can simply download StatusNet and install it, for example on a private server of a company for using it as an internal company blogging service. Identi.ca as a microblogging website is a freely usable public installation of StatusNet. You only have to register with your email address for using the service.

As StatusNet implements the open microblogging standard OStatus[7], different StatusNet installations can communicate and cooperate with each other for building a wide spread distributed blogging network.

[6] `http://indenti.ca`

[7] `http://ostatus.org/`

Figure 11: Identi.ca website.

Though at first it seems logical comparing StatusNet with Twitter, it is not really possible to compare the two different architectures with each other. The biggest StatusNet installation is indenti.ca with only few million users in total, which is way less than what Twitter has to handle.

As we already described in the paper, the evolution in Twitter's architecture and all the connected architectural decisions were based on the scalability problems. Without these problems, Twitter would still be a simple Ruby on Rails website. And that is, what StatusNet is: A simple PHP project.

StatusNet is not designed for endless scaling, as the idea was to develop an open source microblogging tool for more or less private purposes. There are no numbers or benchmarks available about how many users StatusNet is suitable for, but while comparing Twitter's complex approaches for scaling well and the simplicity of StatusNet's PHP project, it seems to be clear that StatusNet is not as scalable as Twitter is. Therefore StatusNet is an alternative when it comes to the available features, but from an architectural point of view, StatusNet is too simple for a comparison.

7.2 Facebook

Facebook is the leading social network and the biggest community website on the Internet. Every user of the around 500 million active users has a profile page, where he can describe himself and share messages, photos, videos and links with his connected friends on the profile's wall. It offers a variety of possibilities for accessing its services, including applications for Android, iPhone, SMS and of course with a regular browser. In total, there are more than 30 billion pieces of content posted every month.

Even though Facebook is much more than a mircoblogging system, by reducing the functionality to posting messages on the wall it is more or less the same as Twitter's Tweets feed. For that reason it makes sense comparing these two systems and in general it is very interesting to compare Facebook's architecture with Twitter's. In some ways they follow the same approach, but other parts of the architecture are really different.

Both of them have to solve the same scalability problem. The dimension of that problem is unique for Facebook and Twitter as there is no other comparable system with that many users and a that dynamic interaction of them.

Facebook is like Twitter based on diversity of open source software and like Twitter Facebook develops and contributes to a lot of open source projects. As Twitter, Facebook is based on a MySQL database which is completely cached in memory by a privately extended version of Memcached. The distributed database Apache Cassandra was developed by Facebook for

the private message inbox search and is used by Twitter as well.

Another analogy between them is that both group a lot of smaller self-developed open source projects around their main architecture for doing exactly that what they really need. Scribe for aggregating log messages or Apache Hive as a data warehouse infrastructure are only two examples of Facebook's open source tools.

The big difference between Facebook and Twitter is how they deliver the memory cached content to the user and backwards. Twitter's approach with Ruby on Rails and a message queueing system in Scala has been described earlier. In contrast to Ruby on Rails, Facebook's website is a PHP implementation and calls itself the second largest PHP website after Yahoo. Instead of developing a message queueing system as Twitter did for speeding up the interaction, Facebook developed HipHop for PHP, a compiler which can transform PHP code into C++ code automatically. Using C++ instead of PHP offers large performance gains, as it works more on a low-level base. By using HipHop, the front end functions can still be written in PHP while the performance advantages of C++ are still given.

Figure 12 gives a brief overview over Facebook's architecture.

Facebook's and Twitter's approaches both work and solve the performance and scalability problems they discovered. Therefore it is a good example to show, that there is no perfect solution when it comes to architectural design. The two companies made their decisions based on the history of their technologies. Twitter was a Ruby on Rails project from the beginning on as Facebook was a PHP project and both still rely on their roots and extend their basic concepts to satisfy their needs.

8 Questions by Other Students

One part of the assignment for this paper was to provide the other students in the software architecture course an article which gives a good overview of Twitter's architectural decisions. Our decision was an interview with some developers of Twitter [11].

In the following each question of the other course members is answered by a paper internal reference or with a few paragraphs, depending on the depth of the question.

Alejandro Serrano Mena

> *"Is the comparing with Scala longer relevant (it seems a bit older). Even more, do you have any insight on why they chose*

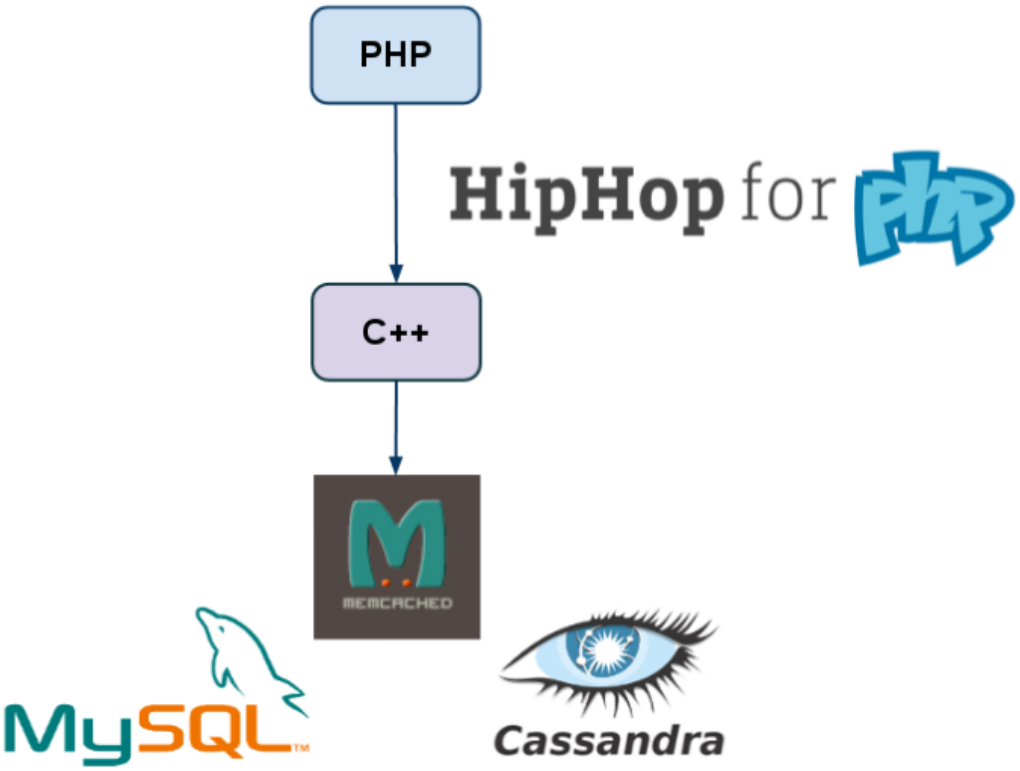

Figure 12: Facebook's raw architecture overview.

Scala instead of Erlang if what they seem to search is improved performance and synchronization and that was already available in Erlang?"

This is still relevant, as they still use Kestrel written in Scala. Their old Ruby on Rails message queuing system was never as good as they wanted it to be, but at the beginning, it was good enough.

Kestrel was already a private side project of Twitter developer Robey Pointer. As a recode of Starling, it was possible to just change one of the message queueing server to Kestrel and investigate, how much faster it was compared to the old Starling servers.

Twitter compared the performance of Kestrel with some other alternatives, but at the end nothing was as fast as the very simple Kestrel with only around 1,200 lines of code.

They couldn't use JRuby without rewriting all of their Ruby gems (packages) that rely on native extensions. Furthermore JRuby interprets Ruby and makes JVM bytecode out of it, but Scala compiles directly to Java bytecode, what the developers prefer. Finally they had no knowledge about how good JRuby performs in such a big project.

The message queue ActiveMQ was much too slow in benchmarks tests.

The famous Message queue RabbitMQ, written in Erlang and therefore the Erlang alternative to Scala based Kestrel was actually Twitter's first choice. Developer Alex Payne said: "I was actually pushing to get us on RabbitMQ, but our benchmarks showed that it just wouldn't work for us, which is a shame, because it advertises some sexy features." The problem was, that RabbitMQ was fast, but with a huge number of messages it ran out of memory an crashed.

Even a meeting with people from RabbitMQ showed, that it is not exactly what Twitter needs.

Another message queue MemcacheQ wasn't already available when they did this evaluation but it's inappropriate due to it's use of Oracle BerkeleyDB, which has an improper copy right licence, even if it is open source.

As a conclusion, Alex Payne said: "We found that Kestrel fits our particular use cases better and more reliably. This was not the hypothesis we had going into those benchmarks, but it's what the data bore out." [2]

Alessandro Vermeulen

"However, can you tell me why they choose Scala? It looks like they did it because it compiles to the JVM, a proven technology

with which they are familiar. Furthermore the JVM has been optimised for long-running processes. Can you tell me why they did not use Erlang or another functional language? They say they've learned to like and love Scala, partly due to it's functional character and 'clean' syntax. However, there are programming languages with a far more functional approach and (in my opinion) an even cleaner syntax."

See the above answer for Alejandro Serrano Mena's question.

Maria Hernandez

"Have you found something about why did they choose Scala instead of other programming languages with similar characteristics, functional and runs on the JVM? such as Clojure or Erjang. If not, have you found any other comparison of Scala with other languages and the reasons for that choice?"

See the above answer for Alejandro Serrano Mena's question. Additional maybe this citation of Twitter's developer Alex Payne helps: "[...] our decision to use Scala over other JVM-hosted languages was based on investigation." [2]

Renato Hijlaard

"In the interview Robey Pointer says that twitter replaced their Ruby based queuing system with one made in Scala.
Why did they deside to rewrite one in Scale while there are many reliable messaging systems around like RabbitMQ? [2]"

Below your mentioned article is a longer discussion with some Twitter developers about exactly that problem. For a more detailed answer see the above answer for Alejandro Serrano Mena's question and the discussion here [2].

Nikos Mytilinos

"Since the article you have suggested contains a discussion about Twitters increased performance upon Ruby on Rails replacement by Scala, I would like to warn you to become more objective in case you are going to use all these arguments in the architecture

*document. Therefore, my actual question is if we sure that Twitter became faster because of Scala displaced Ruby on Rails. For instance, I found this article: [2] that criticise this discussion and suggests that Twitters better performance should be attributed to Scala's strengths, not to Ruby's weaknesses. It would be interesting, thus, to read in your document about the *actual* trade-offs during the architectural redesign of Twitter."*

We are not sure if you can call the switch from Starling to Kestrel really an architectural redesign, as they only replaced one message queueing system with another. Furthermore we are not sure about what you mean with the "actual trade-offs". As mentioned in the interview, they didn't really run into bigger problems. You can find more detailed information about the integration of Kestrel into the system in the above answer for Alejandro Serrano Mena's question and the discussion here [2].

Geert Wirken

"Twitter prefers Scala over Java or other more mature architectures. Based on the article you sent for review, it looks like they chose for Scala because it's flexible (compared to Java and C) and it is more reliable than Ruby, what they used first. However, is there a trade-off between the flexibility provided by Scala and it's performance and reliability? I.e., is Scala just as stable and reliable as Java or C?"

As far as the developers explain that in the given interview [11], scala had no real trade-offs and they even said, that the switch to Scala "worked remarkably well" for them. Reliability and performance were the reasons for choosing Scala. If there is a trade-off connected with introducing Scala, then it could be the maintainability which becomes worth by adding a new language to the company's portfolio.

Jacek Marek

"There was a questions about trade-offs of using Scala. The given answer did not actually provide us with any significant trade-offs of using this language. Have you found any information about Scala trade-offs at all or do you consider Scala to be a trade-offs free language?"

Talking about trade-offs in general is not really possible, because they are usually connected to a specific problem. A fork in general is a good tool for eating, but for soup a spoon would be the better choice.

Scala is for sure not free of trade-offs in some use cases, but it seems like it has been a perfect trade-off free choice for Twitter.

Vladimir Smatanik

> *"Change from Ruby on Rails to Scala was caused by the performance issues of large number of tweets, or by issues with following and un-following of users as well? Are there any known information, how does this particular things work? (in the article was mentioned, that it's done asynchronously by deamons)"*

The reason for changing from ruby on rails to Scala had to do with the enormous amount of tweets that were posted. The message queue called Starling written in Ruby-on-Rails could not handle this large amount and had to be rewritten. This is done in Scala. Deamons take the tweet from the message queue and move it through the system till the tweet is at his location (e.g. on the website or on a mobile phone). The old deamons performed only one action at a time (like notifying a user that there is a new tweet or sending to external device). Nowadays deamons do combined jobs and this led to a big performance improvement. These daemons find their way through the social network by the Flock database. This database keeps track of who is friends with who and is stored in a distributed databse called Gizzard. [1, 3]

Richard Derer

> *"Could you explain a bit the functionality and usage of daemons? We saw a bit about it in the presentations, but still it's a bit unclear for ones, who haven't worked with it yet."*

The original goal of Twitter's daemons was to improve response and processing time. This was achieved by indexing and linking tweets/hashtags/groups in the background. Basically the heavy queue tasks were put into a messaging system which were processed by the daemons. The user did not notice these background processes.

More information on deamons in the answer of the question from Vladimir Smatanik.

Hans Peersman

> *"It is still not clear for me what the daemon exactly does. How does it works?"*

See the above answer for Richard Derer's question.

Alexandru Dimitriu

> *"Twitter not only allows its user to search, but to save the result of search. It's fast growing site, have you found any specific information about issues with searching and other functionality than tweeting?"*

As mentioned in the chapter about Twitter's architecture, Twitter introduced the Apache Lucene library for providing inverted index search for speeding up its search engine. Furthermore Twitter puts for example a lot of effort in making all its features available for API calls in a huge variety of languages, which could also be called as another issue than tweeting.

Mark Rouhof

> *"What is the exact difference between the Scala and Ruby/Python build environment or how does this compile-deploy cycle looks like?"*

Scala has the same well known compile/deploy characteristics as Java. The Scala code has to be compiled to Scala bytecode and runs on a virtual machine. Depending on the purpose, the bytecode has to be used by, for example a Tomcat server for providing website.

In contrast to that, Ruby is an interpreted language which is in most cases interpreted by the Rails webapplication framework. Python is interpreted as well and the standard interpreter for it is CPython, written in C.

Ruben van Vliet

> *"Twitter first used the Ruby on Rails framework for their front and back-end. By switching to Scala, Twitters performance increased. Could you explain what influences this switch has on the old architecture, compared to the new one? And if these changes are responsible for the increased performance, or is the performance of Scala better in general? "*

Kestrel is a Scala recode of Starling. They only switched from servers running Starling to server with Kestrel and didn't change the rest of the architecture. Therefore the performance gain is only based on the advantages of Scala.

Jeroen van der Velden

> *"What is the downside due to a lack of Textmate support and will this improve when Textmate 2 is being launched?"*

Textmate is a Mac OS X text editor, therefore the only downside is a worse coding experience in that operating system. As Apple doesn't give a lot information about Textmate 2 and its new features, we reference to this article [7] which gives a brief introduction about how to use Textmate for Scala development.

Stijn van Drongelen

> *"I would think that using Scala for the front-end instead of Ruby on Rails would bring a performance gain (functional languages are well optimizable), without losing any maintainability (Scala is considered high-level by the developers). Did the developers consider using Scala for the front-end?*
> *If so, why did they decide not to do so? The only obstacle that I can think of would be the loss of the functionality that comes with Rails."*

Twitter has always been a Ruby on Rails website and it worked great. For performance reasons they only had to change the message queuing systems and other back end things like integrating Lucene. There was no need for using Scala in the front end as well and.

Robert Vroon

> *"Twitter is growing very fast and there are more and more apps for Twitter. How many users and apps can Twitter handle at this time? The downtime is less than 2 years ago but how future proof is it?"*

There are no figures around which indicate the amount of possible users with the current architecture.

Theodoros Polychniatis

> *"What do you think of the maintainability of a project that is based on different languages and especially on scala which is very uncommon, however good it may be?"*

Of course this usually reduces the maintainability but on the other side, Kestrel in its basic version has only a few hundred lines of code. The interfaces are still the same as with the earlier Ruby message queueing system. The lack of maintainability is therefore very small.

Thijs Alkemade

> *"Twitter: Being closed-source makes it hard to make this judgement, but do you think it will be advantageous to fully switch to Scala for all of Twitter's back-end? Twitter's use is still growing, and there seems to be a lot to win in terms of performance by switching."*

By looking at Twitter's back end, you can see that the message queue in Scala is used for the whole connection between the database and the front end. Therefore it is more or less already used for the whole back end.

Johannes Leupolz

> *"What is the average time a message needs to pass the internal messaging queue?*
> *Messages in the internal messaging queue should not be lost. How does Twitter ensure fault-tolerance in the messaging queue?"*

We found no figures about the time in the queue or a formal proof about the queue's functionality. As mentioned, Kestrel is only a few hundred lines of code and assuring that the messages can't be lost in that queue should be very easy.

Kevin van Blokland

> *"One important fact in the article on the website is that the Twitter middle layer now consists of Scala applications, handling the api requests. The former implementation was done by Ruby applications, therefore the communication consisted largely (without regarding the public api) of Ruby to Ruby calls. Now this pattern has changed to Ruby to Scala calls. Do you think that*

there is an overhead in casting the api requests to a way Scala can handle them, or is this one of the powerful features of Scala?"

Kestrel and its predecessor Starling both uses the memcache protocol for communications with their clients. With that interface, the switch to the Scala based Kestrel didn't affect that.

Sjors Otten

"Question: Regarding twitter, they switched from ruby on rails to Scala for their back-end processes for performance improvement. They also state that Ruby on Rails and Scala complement each other but it wasnt clear on what level they complemented each other. Could you elaborate / investigate on which particular parts of twitter and to what extend Ruby on Rails is complemented by Scala or vica versa and what the trade-offs are when using two languages/systems?"

Scala combines object-oriented programming and functional programming and runs on a virtual machine like Java. Ruby instead is a dynamic and reflective language which has to be interpreted by, for example the Rails framework. In the article, they want to explain with 'complementing', that Ruby on Rails performs better at front end jobs and Scala better in back end jobs. For the trade-offs with introducing the new language, see the answers for Kevin van Blokland and Jacek Marek.

Lambert Veerman

"The SCALA language makes use of actors to provide concurrency. Alex Payne says in the interview: "For example, it uses a memcache library that Robey wrote, which is actor based. But for other parts weve just gone back to a traditional Java threading model."
What are the tradeoffs for using the SCALA actors instead of the Java threads?"

Twitter gives no more detailed information about that as mentioned in the interview. There they mention that "the engineer working on that, John Kalucki, just found it was a little bit easier to test, a bit more predictable". The negative side effect is that Java and Scala code are mixed in the same project, which reduces maintainability a bit.

Sander van der Rijnst

> *"It looks like twitter took a big risk choosing scala since its not a proven technology. Support resources for scala are lacking and it does not have the backing of an enterprise. This sound great for hobbyists, but it is not really ready for an enterprise. With the success of twitter, do you think this will have a positive effect on Scala as well?"*

Twitter's decision definitely pushed the popularity of Scala. Even if a lot of people complained about that decision, Scala was in the developer news for a while and there are for sure people who started to use Scala after reading all the advantages Twitter discovered with it.

Kevin van Ingen

> *"Twitter has made good progress in picking faster/better performing back-end systems. What are likely front-end replacements for the (near) future?"*

As Ruby on Rails is famous for being one of the most innovative open source communities and still state of the art, it is hard to say if there will be a better solutions for the front end programming. HTML 5 is a big topic in the current web developer discussions, but even that has already been integrated into the Ruby on Rails standard templates.

9 Conclusion

In this paper, we discussed the software architecture of the famous microblogging service Twitter.

We saw, that the biggest problem they had to solve was the lack of performance they discovered, while the amount of users increased dramatically.

The example of Facebook shows, that assuring scalability is a very important challenge for today's big global players on the dynamic social network and microblogging market.

The paper explained how Twitter's architecture changed during Twitter's evolution and how this evolution was connected to the growth of Twitter. Usually these decisions are strongly connected to trade-offs, but we showed that in Twitter's case only the maintainability decreased a little bit. The reason for not having too big trade-offs in the system has to with the succes of Twitter. This succes enables twitter to raise more and more funds to

invest money in to the system and this makes it possible to remove particular important trade-offs. Since 2007 Twitter has raised 360 million US dollars from ventures capatalists and this enables them to remove a lot of trade-offs. All this money can be put in to employees and hardware, the payroll of Twitter is trippled since the beginning of 2010. Currently more than 350 people are working for Twitter. In december 2010 they raised 200 million dollars to invest and at that time Twitter was valued more then 3,7 billion US dollars. All these investments make it possible to keep trade-offs as low as possible. [10]

Furthermore the chapter with the students' questions shows, that not all the topics and facts were clear to everybody. For understanding all of the architectural decisions that Twitter made in the last years, a deeper insight into the backgrounds of the used technologies and concepts is needed.

As a final result, it is very important to underline that scalability is already a big issue for web applications, but as the Internet is still growing, it will also be important in the future. Therefore system architects of the future's web applications will always have to care about scalability and how to provide a high-performance software with as less trade-off problems as possible.

References

[1] Presentation about Twitters components. `http://www.youtube.com/watch?v=_7KdeUIvlvw`.

[2] Tony Arcieri. Twitter: blaming Ruby for their mistakes? `http://www.unlimitednovelty.com/2009/04/twitter-blaming-ruby-for-their-mistakes.html`.

[3] Twitter engineering Blog. Flock DB & Gizzard. `http://engineering.twitter.com/2010/04/introducing-gizzard-framework-for.html`.

[4] International Organization for Standardization. ISO/IEC 9126-1:2001. `http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749`.

[5] Francisca Losavio, Central University of Venezuela, Caracas, Venezuela. Quality Models to Design Software Architecture. `http://www.jot.fm/issues/issue_2002_09/article4/`.

[6] Alexa Internet. Twitter.com. `http://www.alexa.com/siteinfo/twitter.com`.

[7] Mads Hartmann Jensen. Using TextMate for Scala Development. http://www.sidewayscoding.com/2010/08/using-textmate-for-scala-development.html.

[8] Steve Johnson. How Twitter Will Change the Way We Live. http://www.time.com/time/business/article/0,8599,1902604,00.html.

[9] Philippe Kruchten. *The Rational Unified Process: An Introduction (3rd Edition)*. Addison-Wesley Professional, 3 edition, 12 2003.

[10] The Associated Press. Twitter picks up another $200M from investors led by Kleiner Perkins, adds 2 board members. http://ca.finance.yahoo.com/news/Twitter-picks-another-200M-capress-53014316.html?x=0.

[11] Bill Venners. Twitter on Scala. A Conversation with Steve Jenson, Alex Payne, and Robey Pointer. http://www.artima.com/scalazine/articles/twitter_on_scala.html.