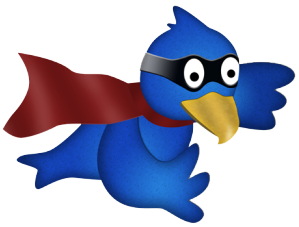
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Specification and Implementation of the Twimight Server

Semester Thesis

Raphael Seebacher
raphasee@ee.ethz.ch

Communication Systems Group
Computer Engineering and Networks Laboratory
ETH Zürich



Supervisors:
Theus Hossmann, Dominik Schatzmann
Prof. Dr. Bernhard Plattner

February 3, 2012

Abstract

As recent natural disasters have shown, microblogging services, such as Twitter, have more and more become a heavily used medium for spreading and acquiring disaster-relevant information. This development has particularly been favoured by the increasing availability of smartphones. However, the availability of fixed-infrastructure networks, which might very well be disrupted as a consequence of a disaster, is crucial for smartphone communication.

In order to reduce the dependency on fixed-infrastructure networks for communication during disasters *Twimight, the mighty Twitter client* for the Android Operating System, has been implemented. Twimight stands out from the mass of Twitter clients in that it additionally provides a *disaster mode* that relies on opportunistic Bluetooth communication, rather than only on the common mode that solely uses fixed-infrastructure networks. Opportunistic communication, as implemented in Twimight, raises various problems compared to the common mode: Apart from the question of how to route and flood packets in the opportunistic network and how to minimize energy consumption of a given mobile device, we identify the role of confidentiality, integrity and authenticity as being a fundamental problem, due to the absence of any connection to the trusted Twitter API.

A *hybrid solution* to this problem has been proposed in [12]. The approach is tagged hybrid since it relies on certain initialization steps to be completed by each Twimight client before the actual diaster happens, i.e., it depends on fixed-infrastructure networks, and is, hence, not purely opportunistic.

This semester thesis realizes the aforementioned hybrid solution. It therefore provides the specification and implementation of the *Twimight Server*, which consists of a web interface and of an *application programming interface* (API). Using this API, the Twimight Server acts as a certificate authority, issuing certificates to Twimight clients. Furthermore the API is designed such as to be extendable for future needs.

Contents

List of Figures

List of Tables

Old programmers never die.
They just can't C as well.

Anonymous

Chapter 1
Introduction

In the past years various smaller and larger natural disasters, like for example the Japan earthquake in March 2011, have hit the world. In such a situation the ability to communicate is regarded as a crucial factor in order to find community, organize rescues, survive until professional forces have arrived and to support disaster relief.

The combination of the popularity of social networks and the availability of smartphones has improved communication capabilities of people during disasters, especially in stages 3 to 5 of a disaster (cf. table 1.1). However, despite the usefulness of this sort of communication, it has a very fundamental drawback. Currently, this communication requires a fixed-infrastructure network, typically cellular networks of mobile phone providers, to be available. In a disaster this fixed-infrastructure is prone to damage, which will eventually result in network outages and disruption of services. Thus, all the advantages from combining social networks and smartphones are on the verge of being lost, since the disruption of fixed-infrastructure networks is quite likely in a disaster case.

A solution to circumvent this disadvantage is provided by *Twimight*, the mighty Twitter client for the Android operating system. Rather than only supporting common communication over fixed-infrastructure cellular networks, or over WiFi, Twimight additionally implements a disaster mode that is based on opportunistic Bluetooth communication. Twimight clients in disaster mode, hence, establish delay tolerant networks, which are current topics in research.

Still, Twimight lacks one very important feature: security. While during normal operation the Twitter API provides security mechanisms, there is no connection to the Twitter API in disaster mode and consequently a distributed solution to the problem should be implemented. Unfortunately, reliable distributed solutions in delay tolerant networks are very difficult to implement and therefore a *hybrid security architecture* has been proposed in [12].

The proposed architecture relies on fixed-infrastructure communication for certain setup steps and not only on opportunistic communication and is therefore

Stage 0 State of social system preceding point of impact	Pre-Disaster
Stage 1 Precautionary activity includes consultation with members of own social network	Warning
Stage 2 Perception of change of conditions that prompts survival action	Threat
Stage 3 Stage of "holding on" where recognition shifts from individual to community affect and involvement	Impact
Stage 4 Individual takes stock, and begins to move into a collective inventory of what happened	Inventory
Stage 5 Spontaneous, local, unorganized extrication and first aid; some preventive measures	Rescue
Stage 6 Organized and professional relief arrive; medical care, preventive and security measures present	Remedy
Stage 7 Individual rehabilitation and readjustment; community restoration of property; organizational preventive measures against recurrence; community evaluation	Recovery

Table 1.1: The eight socio-temporal stages of a disaster. (Source: [7, 25])

called hybrid. An essential part of the architecture is the so-called *Twimight Server*, which is further specified as well as implemented in the course of this thesis.

As specified, the *Twimight Server* provides two separate interfaces, an API for connecting to Twimight clients, i.e. to the Android app, and a web interface for data retrieval and deletion, as well as for certificate revocation. Primarily the Twimight Server acts as a *certificate authority* and issues certificates for Twimight users and distributes the certificates among the Twimight clients. In addition, it maintains and provides the *certificate revocation list*. Furthermore the Twimight Server allows Twimight clients to store location reports in his database and supports clients by providing information about neighboring clients. For the implememtation of the Twimight Server we relied on a combination of PHP, MySQL, OpenSSL and Lighttpd and chose a MVC approach for the actual application.

Even though the Twimight Server is implemented and working as specified, remains just a basic building block for helping in disaster relief. However, many ideas for future development exist and are presented as a conclusion of this thesis.

1.1 Outline of the Thesis

In chapter 2 background information on Twitter, its use during disasters, Twimight, including delay tolerant networks, as well as on web technologies is provided. Then, in chapter 3, the Twimight Server API as well as its web interface is specified. We illustrate the actual implementation of the Twimight Server in chapter 4. Finally, we conclude this thesis in chapter 5 by offering thoughts on how to further develop the Twimight Server.

1.2 Motivation

The apparent utilization of Twimight and, hence, the Twimight Server in disaster cases proved to be a highly motivational factor, i.e., noting that what we develop in this thesis might actually be of direct use and is not just a pure academic issue. Even though writing this thesis finally turned out to be quite a challenge, mainly due to the various technologies involved, the benefit from learning all those technologies, programming languages et cetera will, hopefully, but almost certainly, prove to be valuable for future projects.

1.3 Acknowledgements

I would like to first of all thank *Prof. Dr. Bernhard Plattner* for having given me the opportunity of writing this semester thesis at the Communication Systems Group.
Furthermore I do and did greatly appreciate the continuous support throughout the duration of the thesis and all the valuable inputs I had been given by *Theus Hossmann* and *Dominik Schatzmann*, by email or during one of the various meetings, and I am indeed really grateful for that.
Last but not least I want to thank *Lorenz Koestler* for his helpful inputs regarding PHP programming, MySQL queries and optimization, as well as design patterns in general.

Software and cathedrals are much the same.
First we build them, then we pray.
Samuel T. Redwine Jr.

Chapter 2
Background

In this chapter we provide various background information related to this thesis: Insights into the microblogging service Twitter, especially its increasing usage during disasters in section 2.1 and information about the Twimight Twitter client, with a focus on delay tolerant networks in section 2.2. Twimight Server concepts, as specified in [12], are summarized in section 2.3. Furthermore an overview of current web technologies is given in section 2.4, since the Twimight Server implementation is based on these. Finally, this chapter is rounded off with considerations on security in section 2.5.

2.1 Twitter

The microblogging service *Twitter* was founded in March 2006 and quickly gained worldwide popularity. At its core are the so-called *Tweets*, messages of 140 characters at most, published by Twitter users. Twitter is used by individuals, organisations and companies alike. A comparison between Twitter and other social networks in terms of active users is provided in fig. 2.1.

In order to view Tweets, users can follow each other. User A following user B is said to be a *follower* of user B, whereas user B is referred to as *friend* of user A. The set of Tweets sent by a user's followers is displayed on the respective user's timeline. Furthermore direct messages can be sent to and only read by a specific friend of a given user.

Today Twitter can be accessed using various platforms, such as personal computers, tablets, smartphones, and with custom applications developed for those platforms, not only with the offical Twitter clients. This is due to the *Twitter API*, which provides full access to a users data. Applications using the API have to be registered on Twitter and have to ask each user for the right to use his data. This authorization process uses OAuth[1].

[1] `http://oauth.net`

For further general information on Twitter we refer the reader to `https://twitter.com/about`, for information regarding the Twitter API the reader may have a look at `https://dev.twitter.com`.

2.1.1 The Usage of Twitter during Disasters

Because of the widespread use of smartphones and the simplicity and brevity of Tweets, Twitter became utterly important for communication during disasters. Consequently, researchers started to focus on the usage of Twitter, and other social networks, during disasters. Below we provide a sample of papers that focus on various aspects of Twitter's use in the event of a disaster:

- Research focusing on the usefulness of Twitter for emergency response can be found in [20].
- Whether we can trust what we retweet in a crisis has been studied in [19].
- Analysis of Twitter usage in emergency events compared to normal usage is provided in [14].
- [1] focuses on facilitation of collaborative work during disasters by using Twitter.
- How to find community through information and communication technology during disasters is addressed [25].

For illustrative purposes about how Twitter has been used during disasters, we provide fig. 2.2 and fig. 2.3, which illustrate Twitter usage in the aftermath of the 2011 earthquake in Japan, a disaster with immense consequences for the country.

2.2 Twimight

Twimight, the mighty Twitter client for the Android Operating System, stands out from the mass of Twitter clients in that it provides two modes of communication:

Normal Mode
In its normal mode Twimight essentially provides many things every Twitter client does, like displaying the timeline, sending tweets, retweeting, etc. Twimight uses fixed-infrastructure networks, either cellular network or, if available, Wireless LAN.

Disaster Mode
In the disaster mode, Twimight provides the same basic Twitter functionality as described above, but now relies on opportunistic communication. Furthermore

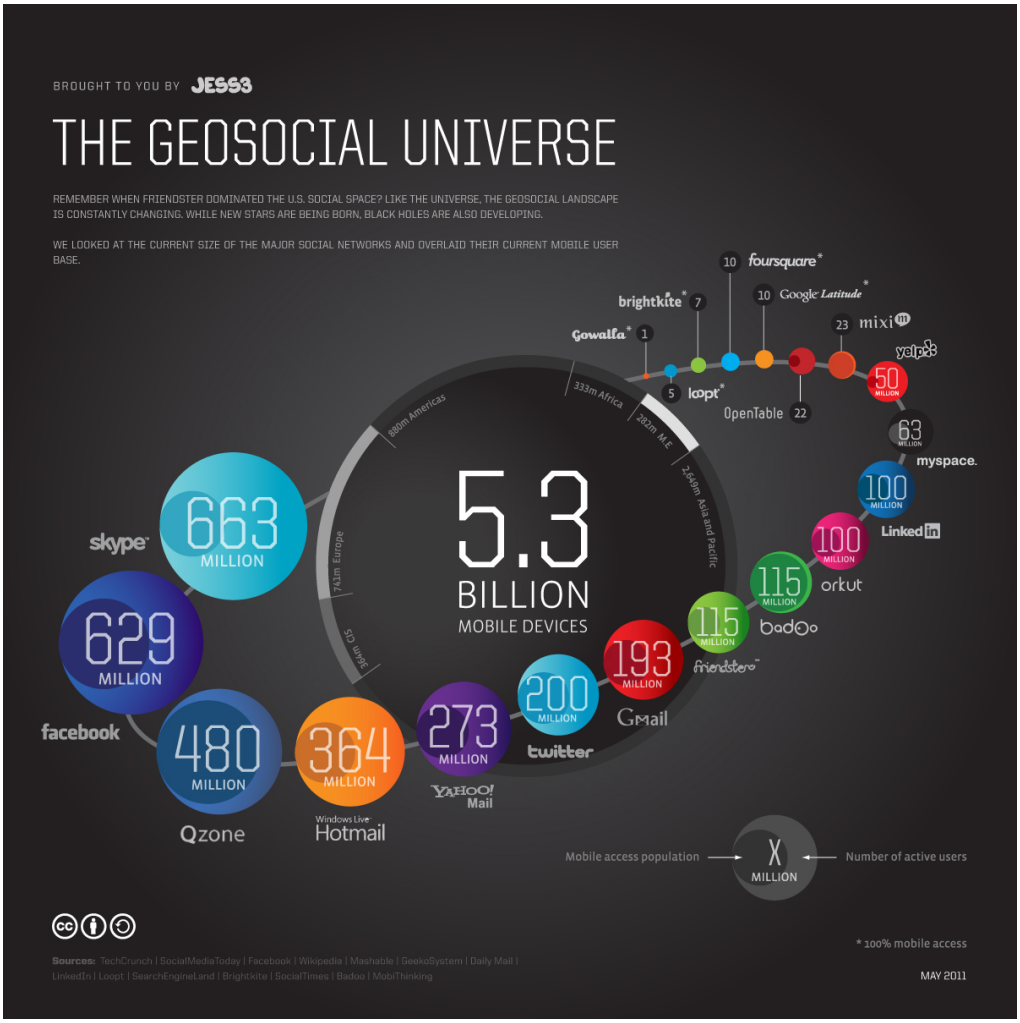


Figure 2.1: Comparison of different social networks regarding their amount of active users. (Source: [16])

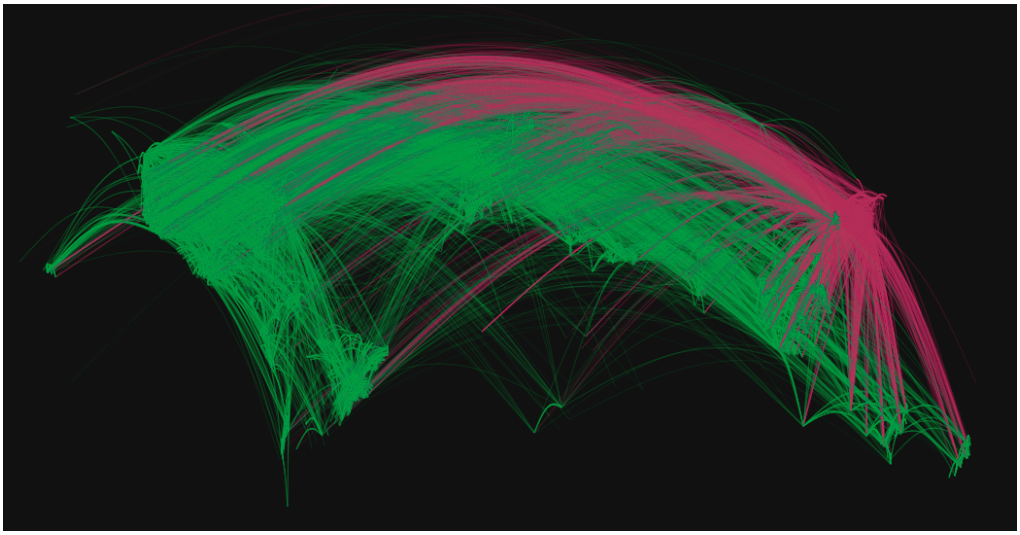


Figure 2.2: Tweets and Retweets from Japan after the 2011 earthquake: Tweets from senders in Japan are shown in red; Retweets by their followers in green. (Source: [5])

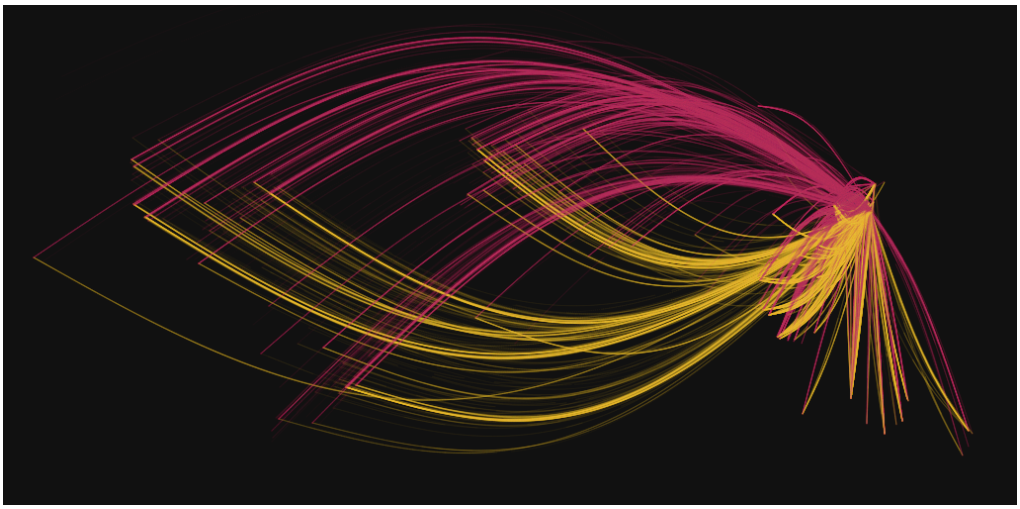


Figure 2.3: Personal messages after the 2011 Japan earthquake: Replies directed to users in Japan are shown in pink; messages directed at others from Japan are shown in yellow. (Source: [5])

tweets are now marked as *disaster tweets* and are epidemically flooded through the network. Disaster tweets are highlighted in red and displayed on every device in that opportunistic network, regardless of follower relations, which helps to greatly improve disaster awareness. Upon returning to normal mode, all disaster tweets are published to Twitter only by their author to prevent duplicates from being published.

For further details on Twimight see [4] and [12].

2.2.1 Delay Tolerant Networks

The disaster mode, which was introduced by Twimight, establishes what is commonly referred to as a *delay tolerant network* (abbr. *DTN*). Their principal goal is to establish a network between a set of nodes without having to rely on any infrastructure. These networks further distinguish themselves from well-known approaches in that they do *not* require permanent connection between all nodes. Due to this lack of continuous end-to-end paths common routing protocol fail to establish routes. Routing in DTNs is researched in [15].

Examples of delay tolerant networks are, amongst others, PodNet[2], Haggle[3], SARAH[4] and Bytewalla[5].

2.3 Twimight Server

So far Twimight has communication capabilities for use in disaster, i.e., its disaster mode. However, in the disaster mode we observe that we cannot provide authenticity and confidentiality due to the lack of connectivity to the Twitter API. To mitigate this problem a *security architecture* has been proposed in [12]. The *hybrid architecture* described therein relies on fixed-infrastructure, i.e., on the *Twimight Server* and its API, as well as the Twitter API. The Twimight Server essentially serves as a *certificate authority* and *public key infrastructure*, but also provides additional services. Authentication of clients is achieved using OAuth and the Twitter API. Thus, the Twimight Server does need to implement an additional authentication processes. For performance reasons client credentials, i.e., the `access token` and the corresponding `secret`, are cached.

In the hybrid architecture each client generates a private key and a certificate signing request (abbr. CSR). The CSR is then sent to the Twimight Server, which signs the CSR with its private key and returns the certificate to the client. The root certificate of the Twimight Server corresponding to its private key is embedded in the Twimight client application. This in turn enables every Twimight client to verify certificates and signatures of messages and tweets, as

[2] `http://podnet.ee.ethz.ch`
[3] `http://haggleproject.org`
[4] `http://www-valoria.univ-ubs.fr/SARAH`
[5] `http://www.tslab.ssvl.kth.se/csd/projects/092106/`

well as to encrypt and decrypt direct messages. The Twimight Server is also responsible for the distribution of certificates and furthermore maintains and issues a *certificate revocation list* (abbr. *CRL*). Mainly for explicit certificate revocation, but also for reviewing the data stored for a specific user the Twimight Server provides a web interface. A sketch of the functionality described above is given in fig. 2.4.

2.3.1 Further Twimight Server Services

To further enrich services of the Twimight Server is extended by the two following features:

On the one hand the Twimight Server enables Twimight clients to store location reports on the server. We define a location report as the 5-tuple (latitude, longitude, timestamp, location provider, accuracy).

On the other hand, the Bluetooth pairing process could be speeded up, if the MAC address of the other device is known, which consequently results in less energy having to be used. In order to do so, we allow Twimight clients to cache their MAC address on the Twimight Server. We then use the location reports that were stored on the Twimight Server to provide the client with a list of MAC addresses of potential neighbors, which can then be used for making Bluetooth pairing more efficient.

2.4 Web Technologies

In this section we give a brief overview of selected web technologies that have been used for this thesis. Note that this chapter is deliberately held short and the reader is given pointers for acquiring further knowledge after the description of each technology.

2.4.1 Hypertext Transfer Protocol and HTTP over SSL/TLS

The *Hypertext Transfer Protocol* (cf. [8]) is an application layer protocol to transport data over a network, which is mainly used in today's world wide web. Communication between client and server is stateless. A message sent from client to server is referred to as a *request*, upon which the server replies with a *response*. Such a message consists of two parts, a *message header* and a *message body*.

HTTP over SSL/TLS (HTTPS, cf. [24]) results from stacking the application layer protocol HTTP on top of the transport layer protocol SSL/TLS, which provides a secure channel. Syntactically HTTPS does not differ from HTTP.

2.4.2 OpenSSL

OpenSSL is an open source implementation of the Secure Socket Layer/Transport Layer Security (SSL/TLS) protocols. It is implemented in C and provides various

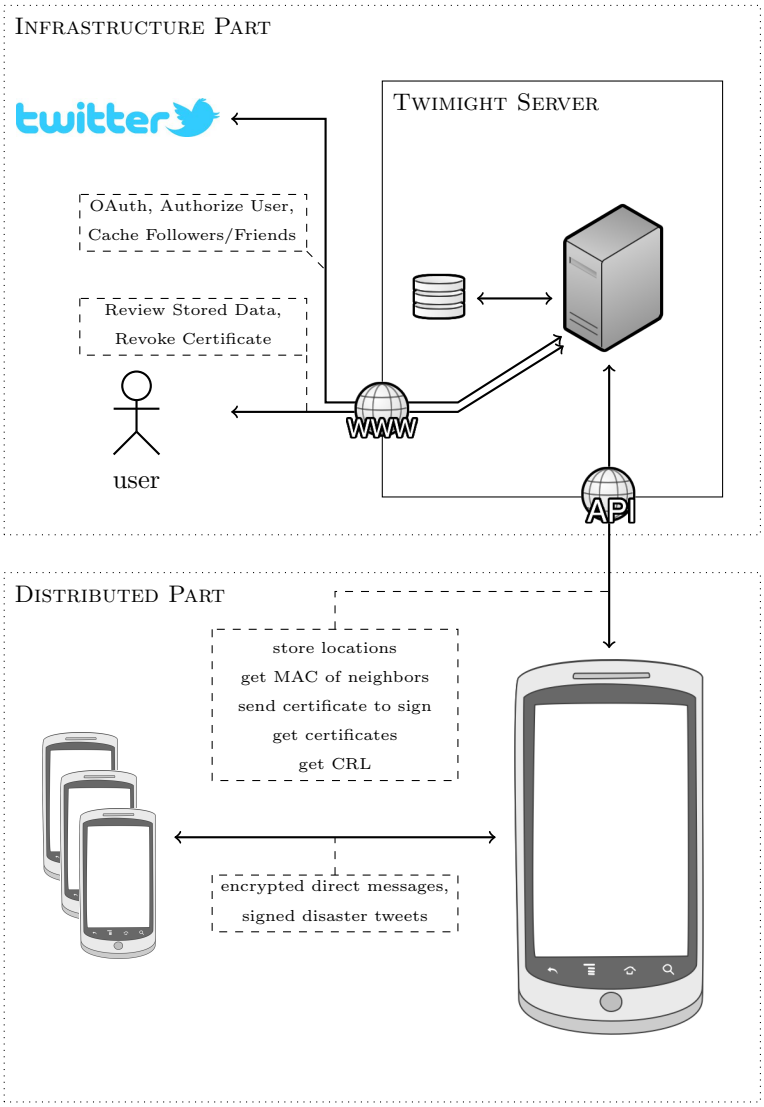


Figure 2.4: Twimight Server operation (Source: [12])

cryptographic functions, as well as a commandline interface, that can be used to request, issue, revoke certificates. More information on OpenSSL, as well as the documentation can be obtained on the official homepage `http://openssl.org`.

2.4.3 Lighttpd

Lighttpd (pronounced *lighty*) is a free, open-source web server optimized for speed and scalability. Originally, Jan Kneschke, the developer of Lighttpd, implemented the web server as a proof of concept for the C10K-problem[6]. However his implementation quickly gained worldwide popularity. Today, various major websites, such as Youtube, Wikipedia or SourceForge, make use of Lighttpd to provide their services.

Compared with the popular *Apache*[7] web server, Lighttpd runs in a single process and therefore is able to answer queries faster while minimizing CPU load and its memory footprint. Benchmarks comparing Apache and Lighttpd show that the latter scales much better for a large number of requests, while the difference between the two is negligible under non-heavy load[8].
The official homepage of Lighttpd can be found at `http://lighttpd.net`.

2.4.4 PHP

First published as *Personal Home Page Tools* in 1995, and later renamed to *PHP: Hypertext Preprocessor*, PHP is today the most widely used server side scripting language[9]. PHP is implemented in C and its syntax has been influenced by Perl as well as C. Recent versions of PHP provide support for object-oriented programming. Furthermore PHP comes with support for various databases and libraries (e.g. OpenSSL). More details regarding PHP can be acquired on the official PHP homepage `http://php.net`.

2.4.5 MySQL

MySQL is a very popular, open-source relational database management system that partly implements the SQL standard. It is often used in conjunction with PHP to serve dynamic websites. Popular users of MySQL are, amongst others, YouTube, Google, Flicker and Twitter.

For high-performance applications, MySQL can be run on clusters and provides means of synchronization for multiple MySQL servers, where one server acts as master. For detailed information, documentation as well as downloads please refer to `http://mysql.com`.

[6] `http://www.kegel.com/c10k.html`
[7] `http://httpd.apache.org/`
[8] `http://www.howtoforge.com/benchmark-apache2-vs-lighttpd-images`
[9] `http://w3techs.com/technologies/overview/programming_language/all`

2.4.6 JavaScript Object Notation

The JavaScript Object Notation provides a format to represent arbitrary data, which remains readable for humans, despite its compactness. Furthermore a JSON object is valid JavaScript, but is as such independent of any programming language. Details of the JSON format are given in appendix C, [6], as well as on `http://json.org`.

2.5 Security Considerations

Whenever deploying a publicly accessible service on the internet, one has to consider various threats. Below we give an overview of the most common attack vectors against web services. Further information, including various examples, as well as testing guidelines, can be found under `http://owasp.org`. A general paradigm, bringing everything to a point, can be stated as follows:

> Never - under any circumstances - trust data from the browser.[10]

This emphasizes the importance of sanitizing every single input provided by a legitimate or unlawful user.

2.5.1 Common Attack Vectors

2.5.1.1 Cross-Site Scripting

Attack
Cross-site Scripting (abbr. *XSS*) attacks enable the attacker to inject an arbitrary client-side script into trusted web sites, which are then viewed by other users. Unfortunately this type of vulnerability is often not payed enough attention, since it affects the users of a website, i.e., the clients and not the server itself. However, as the next paragraph shows, the consequences of XSS can be serious.

Consequences
Defacement of websites, data leakage (most notably the session identifier), drive-by download provisioning.

Countermeasures
XSS attacks can usually be mitigated by escaping every input that is supplied by the user, and also by encoding every output that is sent back to the user.

2.5.1.2 Cross-Site Request Forgery

Attack
A *cross-site request forgery* (abbr. *CSRF*, *XSRF*) is referred to, when a malicious

[10] Excerpt from `http://djangobook.com/en/2.0/chapter20/`

website can forge requests to a honest website, by exploiting the trust established between the user's browser and the honest website.

Example
We assume a user is logged in on `https://twitter.com`, and, hence, a session has been established between client and server. The user now surfs to another, seemingly legitimate site. However, this site contains the following piece of HTML code:

```
<img src="https://twitter.com/tweet.php?message=foo" />
```

After the server downloaded the site, it tries to download the above image, using the URL provided in `src`. Since the user has previously logged in to Twitter, the request for this "image" is sent in the context of the established session with Twitter, which causes the request to be executed with the privileges of the legitimate user. In this example, the malicious `img`-tag results in the user tweeting "foo".

Countermeasures
CSRF attacks vectors can be eliminated by including a hidden token in the request. This token is generated, when the server delivers the content of the page Another way is to use CAPTCHAs[11].

2.5.1.3 SQL Injection

Attack
When serving dynamic content, URLs often include parameters. If those parameters are not properly escaped and directly used to query the database, the web application is vulnerable to *SQL injection* (abbr. *SQLi*).

Example
A request to the URL `http://www.example.com/products.php?id=5` causes the following query to the SQL database to be executed on the server:

```
SELECT * FROM products WHERE product_id=5
```

where the parameter 5 was copied directly from the URL to the query. A malicous user can now alter the URL to, for example, `DROP` the entire table:

```
http://www.example.com/products.php?id=1; DROP TABLE products
SELECT * FROM products WHERE product_id=1; DROP TABLE products
```

Consequences
The attacker gains control over the database and can execute arbitrary SQL queries with the privileges of the user the web application connects to the SQL

[11] `http://www.captcha.net`

server with.

Countermeasures
Obviously, the main countermeasure agains SQLi is proper escaping of the user supplied input. Oftentimes, but depending on the programming language, ready-made escaping functions are available. PHP (cf. section 2.4.4) provides the function `mysql_real_escape_string`[12] to escape queries for MySQL databases (cf. section 2.4.5). Furthermore the privileges of the user, the web application connects to the SQL server with, can be restricted to the absolute minimum.

2.5.1.4 Directory Traversal and Forceful Browsing

Attack
The goal of *directory traversal* and *forceful browsing* is to get access to resources, which are not linked on a website, but are nevertheless accessible. This is usually achieved by manipulation of the URL.

Countermeasures
A very restrictive way of eliminating this threat is to whitelist every single URL, using URL rewriting.

[12] `http://www.php.net/manual/en/function.mysql-real-escape-string.php`

Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

Linus Torvalds

Chapter 3

Twimight Server Specification

To specify the Twimight Server, we first look in section 3.1 at the very fundamental building blocks, which we decided to use. In section 3.2 we illustrate the two interfaces we decided to build and then describe each in section 3.3 and section 3.4, respectively.

3.1 Basic Building Blocks the Twimight Server

Naturally, we rely on a Linux distribution as underlying operating system to provide the most basic functionalities. In particular, Ubuntu 11.10 Server has been chosen. We primarily decided to build the Twimight Server using PHP, because we could rely on a certain amount of experience that we had already gathered. The same is true for MySQL. This decision was backed by the popularity of PHP and MySQL being used to provide web services. A rather unusual decision was to use Lighttpd as web server, but, since Lighttpd scales better for many requests, it seemed to be reasonable. Furthermore we chose OpenSSL to provide the cryptographic functionality we required.

3.2 A Two Interface Approach

From the sketch of the Twimight Server shown in fig. 2.4, we identify the two interfaces the server offers: On the one hand we have the API, and on the other hand the web interface. The requirements for the two interfaces are fundamentally different. While the web interface has to guarantee a smooth experience for the user, the API has to be optimized for high throughput and efficiency, since it is accessed rather frequently and in an automated manner by the Twimight clients.

3.3 API Interface Specification

In order to enable a Twimight client to communicate with the Twimight Server, we make use of the *Hypertext Transfer Protocol* over SSL/TLS (cf. section 2.4.1) and of the *JavaScript Object Notation* (JSON) data exchange format (cf. section 2.4.6). We chose JSON as the notation to represent messages, because it has, compared to e.g. XML, the key advantage of remaining human-readable while offering a very succinct representation of data.

Further the API is designed such that all operations it provides can be invoked from one single message. The response then follows in one single message as well. To our opinion this ensures simplicity and efficiency of the API.

Notation 3.1 (Parameter) *We specify a parameter, that has to be supplied, using angle brackets, i.e. in the form* `<parameter>`.

Notation 3.2 (Optional Statements) *If some statement, e.g. a parameter, is optional, it is enclosed in square brackets.*
Example: *A single, optional parameter is represented in the form* `[<parameter>]`, *some arbitrary statement in the form* `[statement]`.

Note Do not mistake optional statements as given in Notation 3.2 for JSON Arrays (cf. fig. C.2). It should be clear from the context to which one of those two we are referring to.

3.3.1 Request Format

Definition 3.3 (Generic Request Format) *We define the format of a generic request by a Twimight client as follows, using the HTTP:*

```
POST <identifier> HTTP/1.1
Host: <host>
Content-Type: application/x-www-form-urlencoded
Content-Length: <content_length>

message={"authentication":<auth_request_object> \
[,"location":<location_request_object>] \
[,"neighbor":<neighbor_request_object>] \
[,"ff_certificates":<ff_certificates_request_object>] \
[,"revocation_list":<revocation_list_request_object>] \
[,"certificate":<certificate_request_object>]}
```

The following parameters have to be provided, in order for the request to be valid syntactically:

- `<identifier>` *is the path to the root folder of the Twimight Server API, possibly* `/api`, *or simply* `/`;

- `<host>` *is the hostname under which the Twimight Server is accessible;*
- `<content_length>`*, which is the length of the content of the request, as defined in [8];*
- `<auth_request_object>`*, structure cf. section 3.3.1.1.*

The parameters

- `<location_request_object>`*, structure cf. section 3.3.1.2;*
- `<neighbor_request_object>`*, structure cf. section 3.3.1.4;*
- `<ff_certificates_request_object>`*, structure cf. section 3.3.1.5;*
- `<revocation_list_request_object>`*, structure cf. section 3.3.1.6;*
- `<certificate_request_object>`*, structure cf. section 3.3.1.7.*

are optional. ◇

Note We refer to each part of the above `message` as a *message element*. For example, `"authentication":<auth_request_object>` is a message element.

A very important property of the definition of the request is, that the API remains extendable. If a new functionality has to be provided through the API, the request can simply be added another message element.

3.3.1.1 Structure of `auth_request_object`

```
{"access_token":<access_token>, \
"access_token_secret":<access_token_secret>, \
"version":<version>}
```

where `<access_token>` and `<access_token_secret>` are the credentials obtained by the Twimight client application during the OAuth authentication process with the Twitter API; and `<version>` is a key to the version of the Twimight credentials used in Twitter API communication, i.e. to the OAuth consumer key.

3.3.1.2 Structure of `location_request_object`

```
[location_report_1, location_report_2, ...]
```

The `location_request_object` is a JSON array of `location_report`s (cf. section 3.3.1.3).

Note The array may be empty, if the client has no `location_report`s that need to be pushed to the server. It is however recommended to not provide the entire `location_request_object` in that case, since the `location_request_object` as such is optional (cf. definition 3.3).

3.3.1.3 location_report

A location_report basically holds information for one location, recorded at a given time, by a specific sensor with a certain accuracy. It has the following structure:

```
{"latitude":<latitude>,"longitude":<longitude>, \
"accuracy":<accurracy>,"timestamp":<timestamp>, \
"provider":<provider>}
```

where

- <latitude> and <longitude> are the coordinates of the location given in degrees, i.e. in the form 47.379022, 8.541001, with

  $$\texttt{<latitude>} \in [-90, 90] \qquad \text{and} \qquad \texttt{<longitude>} \in [-180, 180]$$

- <accuracy> is the accuracy of the location in meters and rounded to one meter as provided by most sensors;
- <timestamp> represents the time when the location was recorded. This timestamp has to be provided in Unix time[1];
- <provider> either equals "gps" or "network", depending on the sensors that were used to obtain that specific location.

3.3.1.4 Structure of neighbor_request_object

```
{["bluetooth":{["mac":<mac_address>]}] \
["wifi":{["mac":<mac_address>]}]}
```

where <mac_address> corresponds to the (wifi or bluetooth) MAC address of the Twimight client that issues the request, formatted as a string, colon-separated, i.e. in the form "01:23:45:67:89:AB".

Note Even though both the "bluetooth" as well as the "wifi" part in the above neighbor_request_object are marked optional, at least one of them has to be provided in a legitimate request. If neither will be provided, the entire statement "neighbor":<neighbor_request_object>], as defined in definition 3.3, has to be omitted.

Note The MAC address need not be provided in every single request, as it is stored on the Twimight Server. When not provided, the Twimight Server uses the last MAC address that has been provided for the specific technology (i.e., for bluetooth or wifi).

[1] http://en.wikipedia.org/wiki/Unix_time

3.3.1.5 Structure of `ff_certificates_request_object`

```
{["last_update":<timestamp>]}
```

where `<timestamp>` represents the time when the follower-and-friends certificates were last updated. This timestamp has to be provided in Unix time[1].

Note If the follower-and-friends certificates have never been updated yet, the `last_update` parameter has to be omitted.

3.3.1.6 Structure of `revocation_list_request_object`

```
{["last_update":<timestamp>]}
```

where `<timestamp>` represents the time when the revocation list was last updated. This timestamp has to be provided in Unix time[1].

Note If the revocation list has never been received yet, the `last_update` parameter has to be omitted.

3.3.1.7 Structure of `certificate_request_object`

```
{"certificate_signing_request":<certificate_signing_request>}
```

where the parameter `<certificate_signing_request>` is a Base64 encoded PKCS#10 (cf. [21]) certificate signing request.

3.3.2 Response Format

Definition 3.4 (Generic Response Format) *Upon a request by a Twimight client, the Twimight Sever replies using the following format, provided that the request matched the generic request format shown above:*

```
HTTP/1.1 200 OK
Content-Type: application/json

{"authentication":<auth_response_object> \
[,"location":<location_response_object>] \
[,"neighbor":<neighbor_response_object>] \
[,"ff_certificates":<ff_certificates_response_object>] \
[,"revocation_list":<revocation_list_response_object>] \
[,"certificate":<cert_response_object>]}
```

The above parameters are present in an arbitrary response, if the associated request has also contained these parameters. ◇

Note Depending on the system environment of the Twimight Server, the HTTP header part might contain more parameters compared to those given in the response above.

3.3.2.1 Structure of `auth_response_object`

Regarding the returned `<auth_response_object>` we have to distinguish three distinct cases:

- If the authentication process with the Twitter API is successful, then

  `<auth_response_object>` = `{"status":"ok"}`

- If the authentication process with the Twitter API fails, then

  ```
  <auth_response_object> =
  {"status":"error","msg":"Authentication failed."}
  ```

- When, according to section 3.3.1.1, not all parameters are provided, then

  `<auth_response_object>` = `error_missing_parameters`

  where the `error_missing_parameters` object is defined in section 3.3.2.7.

3.3.2.2 Structure of `location_response_object`

For the `location_response_object` we have to consider three different cases:

- If all location reports have successfully been parsed and stored, the server replies with

  ```
  location_response_object = {"status":"ok"}
  ```

- If some, but not all location reports have been parsed, the server replies with a warning

  ```
  location_response_object = {"status":"warning", \
  "msg":"Only <lr_successful> out of <lr_total> \
  location reports could be parsed."}
  ```

  where `<lr_successful>` is the amount of sucessfully parsed location reports and `<lr_total>` is the total amount of location reports provided in the request.

- If no location report has been parsed successfully, the server replies with an error

  ```
  location_response_object = {"status":"error", \
  "msg":"No location reports could be parsed."}
  ```

3.3.2.3 Structure of `neighbor_response_object`

The `neighbor_response_object` generally has the following structure

```
{"bluetooth":<neighbors>,"wifi":<neighbors>}
```

where the `"wifi"` and the `"bluetooth"` part are provided if they are present in the associated request.

The parameter `<neighbors>`, provided for each technology seperately, is of the following form:

- `{"status":"ok","mac":<mac_array>}`

  where the parameter `<mac_array>` is a JSON Array of MAC addresses, which are formatted as specified in section 3.3.1.4.

- If no neighbors have been found, then

  ```
  {"status":"error","msg":"No neighbors found."}
  ```

3.3.2.4 Structure of `ff_certificates_response_object`

The `ff_certificates_response_object` is of the following form:

```
{"status":"ok","certificates":<ff_certificates>}
```

The parameter `<ff_certificates>` is a JSON object of the form specified below:

```
{<ff_1_id>:<ff_1_certificate>,[<ff_2_id>:<ff_2_certificate>,...]}
```

where `ff_`i`_id` is the Twitter User Id of the ith follower-and-friend and the parameter `ff_`i`_certificate` contains his current X509 certificate. If no follower-and-friends certificates are stored on the server, we have

```
<ff_certificates> = {}
```

3.3.2.5 Structure of `revocation_list_response_object`

The `revocation_list_response_object` that is returned by the server is of the following form:

```
{"status":"ok","crl":<crl_array>}
```

where `crl_array` is a JSON array consisting of the following entries, each of them representing one revoked certificate:

```
{"expiry":<certificate_expiration_timestamp>, \
"revocation_time":<revocation_time_timestamp>, \
"serial_number":<certificate_serial_number>}
```

3.3.2.6 Structure of `cert_response_object`

If the certificate signing request provided in the associated request was valid, and no errors occurred in the singing process, the following `cert_response_object` is returned:

```
{"status":"ok", \
"signed_certificate":<signed_certificate>}
```

where `<signed_certificate>` is the signed X509 certificate.

If, however, errors occur, the returned `cert_response_object` has the following form:

```
{"status":"error","msg":<message>}
```

where the parameter `<message>` is one of the following, depending on the error that occurred:

- `"Certificate signing rate limited: max 1 \`
  `certificate signings per user per day!"`
- `"CSR has already been signed."`

3.3.2.7 `error_missing_parameters`

If the list of parameters provided for some request object is incomplete, i.e. if it does not match the specification in section 3.3.1, the Twimight Server returns the `error_missing_parameter` object instead of the response object that corresponds to the request:

```
error_missing_parameter = {"status":"error", \
"msg":"Not all mandatory parameters have been provided.", \
"missing_parameters":<missing_parameters_array>}
```

where `<missing_parameters_array>` is a JSON array containing all parameters that have not been provided.

3.4 Web Interface Specification

While the API provides automated access for Twimight clients and, in principle, remains hidden from the user, the web interface is directly accessed by the user and enables him to do the following: Most important, the user can revoke the certificate that has been generated and uploaded by the Twimight client on his Android smartphone. This feature is useful, when for example the user's smartphone was stolen. Furthermore the user can access and assess every piece of data that has been stored on the server and, hence, has full control over his data. Finally the user can delete all his data that is stored on the Twimight Server.

For the authentication process we use OAuth in conjunction with the Twitter API[2]. Thus there is no need to implement a specific login and register feature, and the client never shares his Twitter password with us. Additionally the web interface has to provide basic usability, a minmal design and advanced features.

The result of the web interface implementation is shown in fig. 3.1.

[2]cf. `https://dev.twitter.com/docs/auth/sign-twitter`

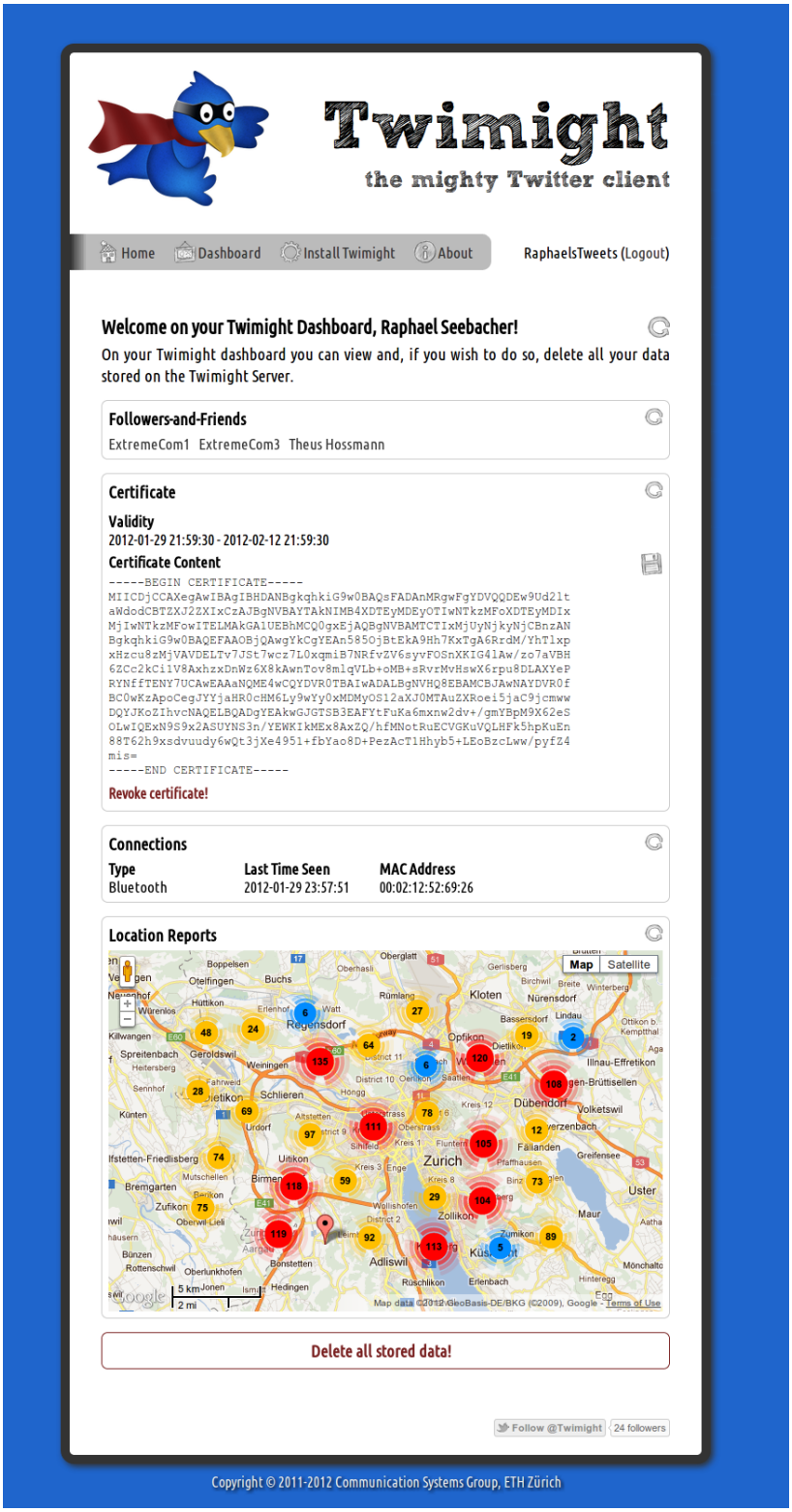


Figure 3.1: The Twimight Server Web Interface

If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident.

Rob Pike

Chapter 4

Twimight Server Implementation

This chapter gives insights into the implementation of the Twimight Server. In section 4.1 we introduce the model-view-controller pattern and then present how we implemented a framework for the Twimight Server based on that pattern in section 4.2. We then show in section 4.3, how we extended that framework to provide the API. Furthermore we present the implementation of the web interface in section 4.4.

We deliberately did not include any source code in this chapter, but point the reader to the Subversion repository[1] to check out the source code for further examination. For information regarding practical development we refer the reader to appendix B.

4.1 The MVC Pattern

Model-view-controller (abbr. *MVC*) is an architectural pattern for structuring software development. It was first described in 1979 by Trygve Reenskaug[2], while working on Smalltalk.

The MVC pattern divides an application into three parts, the *model*, the *view* and the *controller*, each of which has its specific role: The model is responsible for the actual data, and further provides operations that can be applied on that data. The view's responsibility is to render data into a presentable form. This depends on the user interface, which may be a website in a browser, or a commandline interface. The controller implements the actual logic of the application. It therefore processes a request, collects data from models and renderes this data

[1] `svn checkout http://twimight-disaster-server.googlecode.com/svn/trunk/ twimight-server-read-only`

[2] cf. [23]

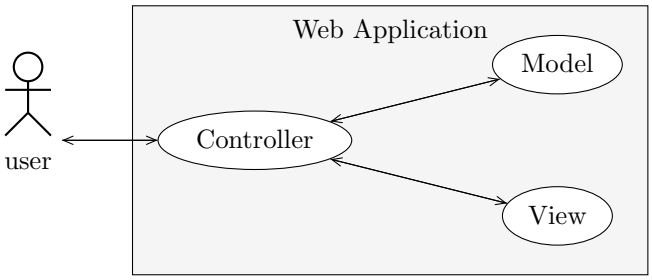


Figure 4.1: The model-view-controller pattern.

using views before responding back to the user. The relation between model, view and controller is shown in fig. 4.1. The benefit from using the MVC pattern is quite obvious: Since the application has to be separated into parts that are orthogonal to each other, unnecessary coupling is removed. This in turn permits independent development, testing and maintenance of each, which we consider especially advantageous. The downside of this approach is certainly to have to invest a considerable amount of time in proper modeling of the application. However, this downside is considered negligible compared to the benefits a MVC approach provides.

Note that there exist many different interpretations of the MVC pattern, in particular regarding the responsibilities of model, view and controller.

4.2 Building the Framework

For scalability and efficiency reasons, and to keep the code as simple as possible, we decided not to rely on a prefabricated framework. Instead, we developed our own lightweight framework completely from scratch. Thereby we followed the MVC pattern, as described in section 4.1, but also provided the necessary surroundings.

4.2.1 Folder Structure

One of the most basic things for the framework was deciding on its folder structure. The structure proposed in [9] served as an inspiration. After slight modifications, we decided on the folder structure depicted in fig. 4.2. Below, we specify what content is stored in which folder, to get an insight at how the application works:

- `application/` contains the actual business logic, hence, the models, the views and the controllers.
- `config/` holds all configuration files, i.e., the web server configuration, the application specific configuration and the OpenSSL configuration files.

```
application/
    controllers/
        message_elements/
    models/
    views/
config/
data/
    certificate_authority/
        certificates/
    logs/
    sessions/
library/
public/
    fonts/
    images/
    javascript/
    stylesheets/
scripts/
    background/
    certificate_authority/
    install/
templates/
tests/
    request_response_tests/
tmp/
```

Figure 4.2: The folder structure of the Twimight Server.

- `data/` stores all application related data. This includes session information, the certificate database, as well as log files.
- `library/` holds libraries from external sources. In particular, the `OAuth` and the `TwitterOAuth` classes are stored here (cf. [27]).
- `public/` is the document root folder as seen from the web server. Consequently all data that is publicly available is stored in this folder. Further the application is invoked by the `index.php` located here.
- `scripts/` contains scripts that are used for either setting up the application, or scripts that are invoked by the application when processing a request, or for cron job scripts.
- `templates/` holds all templates used for serving content, most importantly for the website.
- `tests/` provides a place for test scripts.
- `tmp/` contains temporary data. Its contents may be deleted at any time without having effects on the application itself.

4.2.2 Configuration Files

The next step in building the framework was to write custom configuration files for the web server, i.e., Lighttpd (cf. [3, 17]), for OpenSSL (cf. [22]) and for the application in general. The content of these files can be viewed in the `config/` folder.

Most important to note is the rewrite rule in the `lighttpd.conf` file:

```
url.rewrite-if-not-file = (
    "^/([^.?]*)\?(.*)$" => "/index.php?request=$1&$2",
    "^/([^.?]*)$" => "/index.php?request=$1",
)
```

This regular expressions ensures, that the whole request URL is rewritten to the `index.php` file, which resides in the `public/` folder, with the requested URL provided in the GET parameter named `request`.

4.2.3 MVC Framework

Finally a basic MVC framework had to be implemented. Below we provide a list of all basic models, views and controllers we built, including a small description of their functionalities and tasks. For more detailed information, please refer to the source code itself.

4.2.3.1 Models

- `abstract class model` is the abstract parent class to all models.
- `abstract class db_model` is the abstract parent class to all models, that require a connection to the database. It hence provides common database-related functions.
- `class request extends model` models and abstracts a request as sent, e.g., by a clients browser.
- `class session extends model` models the PHP `$_SESSION` variable.
- `abstract class log extends model` provides an abstract class for all log, independent of how they are saved.
- `class log_text extends log` represents a log saved in a text file.

Exceptions

A special case are the exceptions, which are used to signal failures during execution. Generically we added the following exception classes:

- `class tds_exception extends Exception` is the general exception class thrown within the application, when an error occurs.
- `class model_exception extends tds_exception` is used to signal errors when operating with a model class.

4.2.3.2 Views

- `abstract class view` is the parent class for all views to be implemented in section 4.3 and section 4.4.

4.2.3.3 Controllers

- `abstract class controller` is the abstract parent class to all controllers.
- `class db_controller extends controller` is a controller that reads the configuration file and provides a connection to the database for querying. Implemented using a Singleton design pattern[3].
- `abstract class front_controller extends controller` is the first controller to be invoked. Since it is abstract and cannot be instantiated, it provides a static function that generates and returns a subclass of it, depending on the request, i.e. on the `request` GET parameter. Its subclasses are

[3] cf. [10, p. 127ff]

  - the `api_controller` (cf. section 4.3),
  - the `web_controller` (cf. section 4.4),
  - and the `ajax_controller` (cf. section 4.4).

- `class log_controller extends controller` is a very simple controller to create a log file, that can be used for special logging as well as for debugging.
- `class config extends controller` provides access to the application configuration file.
- `final class autoload` is a very *special case*. Since all classes are related to each other, stored in various files and invoked dynamically during execution, the location of the files, where a given class is stored has to be known. The `autoload` class utilizes a PHP mechanism[4] to load classes automatically.

4.2.4 Design of the Relational Database

The actual design of the tables of the relational MySQL database was straight forward from the `db_models` implemented in section 4.3. Thereby data for each `db_model` is represented in the database by one table. The final structure of the database is provided in appendix D.

From a technological point of view, we choose *InnoDB* as storage engine for all tables. Reasons for choosing InnoDB over *MyISAM* are its support for foreign key constraints in particular and its reliability in general[5]. Furthermore InnoDB supports row-level locking, whereas MyISAM only supports table-level locking.

4.3 API Implementation

We first show what we generally added to the model-view-controller framework to provide the API, and then incrementally show the specific additions for each message element.

4.3.1 General Additions to the MVC Framework

4.3.1.1 Views

- `class json_view extends view` provides a view to render data into a JSON object, as needed for the response (cf. section 3.3.2).
- `class error_view extends view` is a view to return errors to the client. This essentially is an abstraction of HTTP status codes.

[4] cf. `http://www.php.net/manual/de/function.spl-autoload-register.php`
[5] `http://tag1consulting.com/MySQL_Engines_MyISAM_vs_InnoDB`

4.3.1.2 Controllers

- `class api_controller extends front_controller` is an implementation of the `front_controller` to process requests directed to the API. It uses the `request` model to get the `message` and then invokes the corresponding `message_element_controller` for processing each message element seperately. Thus, the `message` is split into its parts, i.e., its message elements and processed by their controller, which is essentially a *divide and conquer* approach.

- `abstract class message_element_controller extends controller` is an abstract class, a prototype for concrete implementations of message element controllers. An implementation of this controller is responsible for processing the information contained in one single message element.

Note Adding a new message element essentially is writing an implementation of the abstract `message_element_controller` class, which is conceptually very straight forward and thereby guaranteeing API extendability.

We furthermore added the following exceptions, in order to signal specific errors that occur while processing a request:

- `class no_element_exception extends tds_exception` is thrown, if a message element is configured as required (cf. `config/config.ini`), but not given in the request.

- `class no_message_exception extends tds_exception` is thrown, if a request to the API does not contain a message, or cannot be decoded.

- `class warning extends tds_exception` is thrown by message element controllers in order to signalize, that the message element could not be fully processed. It contains a human readable string that describes the warning.

- `class error extends tds_exception` is thrown by message element controllers, if a message element could not be processed at all. It contains a human readable string to further describe the error that occurred.

- `class missing_parameters extends error` is thrown, if a message element does not contain all parameters it requires for being processes. It also contains a list of the parameters that are missing.

4.3.2 The `authentication` message element

To be able to authenticate the user, we needed a connection to the Twitter API. We therefore implemented the following two models:

- `abstract class api_connection extends model` provides an abstract parent class for all potential implementations of api connections, even if there is currently only the one to the Twitter API.
- `final class twitter_api_connection extends api_connection` offers specialized functions to connect to the Twitter API. It makes use of the `TwitterOAuth` class, as implemented in [27].

To hold the data we needed, we implemented the models below:

- `class server_credentials extends db_model` basically hold the pair of `consumer_key` and `consumer_secret`, to enable connection to the Twitter API.
- `class user_credentials extends db_model` represents valid credentials of a user, that were obtained from the Twitter API.
- `class user extends db_model` represents a user and functions to get related objects, such as, e.g. his credentials, i.e. an instance of the `user_credentials` class.

We finally added the following two controllers to actually process the message element:

- `final class authentication extends message_element_controller` is very simple: it basically dissects the message element, creates an instance of the `user_credentials` model with the provided credentials and invokes the `user_controller` for authentication.
- `class user_controller extends controller` encapsulates the functionality to authenticate a user, the state, i.e. if the authentication was successful or not.

  Since the Twitter API is quite slow, the following procedure is applied when a user sends a request:

  1. Check if the `user_credentials` provided already exist in the database, if true, then set the corresponding user as authenticated and return.
  2. Check the validity of the supplied credentials using the Twitter API. If successful, then create and save a new user, save the supplied credentials for the newly generated user, set him as authenticated and return.
  3. Return an authentication error.

  This local caching resulted in a drastical increase of performance.

4.3.2.1 Follower-and-Friend Update Script

Since the followers and friends (i.e. the users that are following him) of a users vary over time the Twimight Server needs to keep track of these relations. To do so, we implemented a script that connects to the Twitter API and updates all these relations daily, using a cron job. If, however, a new user joins, the script is invoked directly from the web application and the relations are updated just for the new user. This speeds up the certificate distribution process for the new user, as the new user does not have to wait half a day on average.

For scalability reasons, we decided to only store a relation, when the users mutually follow each other and, hence, its name *follower-and-friend*.

4.3.3 The `certificate` message element

To provide the most central part of the API - the signing of certificate signing requests - we added the following controller to our framework:

```
final class certificate extends message_element_controller
```

Furthermore, we implemented two models, as described below:

- `class certificate_signing_request extends model` contains the certificate signing request as sent by the client and provides functionality to sign the request, which results in a valid, signed certificate.
- `class user_certificate extends db_model` is an abstraction of a X509 certificate and provides functions to, e.g., get its properties, or revoke it.

The functionality of the `certificate` message element controller is fairly simple: He first checks, whether the user still has a valid certificate and revokes that certificate. Then he instantiates a `certificate_signing_request` model and invokes its signing function. Upon success, he retrieves the signed certificate and stores it in the database. He finally returns the signed certificate to the user. To keep the server load limited, we further added a certificate signing rate limitation per user. That is, a user can, e.g., only get one certificate signed per day. However, this limitation does not apply for manually revoked certificates!

The challenge with implementing the *certificate authority* was coping with PHP's limited customizability of OpenSSL. We therefore decided to use the OpenSSL commandline tool for signing requests and for revoking certificates. However, this forced us to use the OpenSSL certificate authority implementation, which relies on files. Therefore, we had to extend the above models to synchronize the file database of OpenSSL with our MySQL database.

4.3.4 The `revocation_list` message element

The implementation of the certificate revocation list message element was straightforward. Only the controller below had to be implemented.

```
final class revocation_list extends message_element_controller
```

The controller asks the `user_certificate` model to give him the certificate revocation list. The `user_certificate` model then provides information of certificates that have been revoked but are not yet expired.

4.3.5 The `ff_certificate` message element

The implementation of the `ff_certificate` message element, which returns the certificates of the authenticated user's follower-and-friends, was simple and did not require much logic. The controller

```
final class ff_certificates extends message_element_controller
```

asks the `user_controller` for the user model instance of the authenticated user and gets an array containing his followers-and-friends. The certificate of each is then collected and returned.

4.3.6 The `location` message element

To enable the Twimight Server to store location reports provided by a client, we added the `class location_report extends db_model` model to our framework. To process the actual message element, the

```
final class location extends message_element_controller
```

message controller was added furthermore. The `location` message controller iterates over all location reports provided in the request, and stores those that contain all necessary data in the database, using the `location_report` model.

4.3.7 The `neighbor` message element

For processing the `neighbor` message element, we needed another model, in order to basically store the MAC addresses of the interfaces of the clients smartphone. We therefore implemented the following model:

```
class user_connection extends db_model
```

To process the message element itself, we implemented the

```
final class neighbor extends message_element_controller
```

controller. The processing of the message element is done in two parts: Foreach of the two available types (`wifi` and `bluetooth`), the following is done if they are given in the request:

1. Update the MAC address of the client in the database, if it is given in the request,

2. Retrieve all neighbors, put their MAC addresses into an array and return that array.

To determine a user's neighbors, we use the most recent location report of that user and the most recent location reports of all other users and then select those users with a location report within a specified distance, using the Haversine formula.

4.4 Web Interface Implementation

4.4.1 Server-side Implementation

To provide the web interface, we extended our MVC framework with two controllers. The `web_controller` serves the whole website synchronously, whereas the `ajax_controller` is responsible for processing asynchronous AJAX requests.

- `class web_controller extends front_controller` basically checks the supplied URL, i.e., the `request` GET parameter, processes the request, instantiates the corresponding view and responds. In special cases, such as the login redirection to Twitter, no view is generated, but rather a HTTP Redirection header is sent back to the client.
- `class ajax_controller extends front_controller` analogously to the `web_controller` it processes requests and returns a JSON object, which is then further parsed in the browser of the client.

Compared to the API, as described above, the *views* for the web interface are much more important, since the response is not just a JSON object, but rather a full-scale HTML page, with cascading stylesheets (CSS) and JavaScript. We therefore first built the `abstract class web_view extends view`, which provides a very basic form of a page, including a small templating engine. It yet is abstract, because to actually make up a page content and a further HTML template is needed. Subclasses of the `web_view` class, i.e.,

- `class home_view extends web_view`,
- `class dashboard_view extends web_view`,
- `class install_twimight_view extends web_view`, and
- `class delete_view extends web_view`

each extend the `web_view` class in that they provide a HTML template of the actual page, as well as a list of stylesheet and JavaScript files that have to be included.

Whenever a request cannot be processed at all, or if an error occurs while processing a request, e.g. if a wrong URL was provided, then the `web_controller` instantiates the `class error_web_view extends error_view` class, which shows an error message to the user. A sample error message is shown in fig. 4.3.

4.4.2 Client-side Implementation

With the server-side implementation provided, as just described, the web interface generally works. However, a web interface today are usually very rich in functionalities. We therefore used various third-party implementations, namely the Google Maps API, jQuery, and various other plugins. We put all these plugins together by writing simple JavaScript code, which can be found in the `public/javascript/` folder.

The most important part of the web interface is the dashboard, which relies on JavaScript to a great extent. The dashboard is the place, where the user can assess all his data on the Twimight Server and take actions, e.g., to revoke the certificate or to delete all his data. The dashboard page as provided by the Twimight Server does not yet contain any content, but rather a JavaScript file in its header (i.e. `public/javascript/dashboard.js`). This script then asynchronously fetches all data from the server, and puts it onto the dashboard page. This greatly simplified and modularized the server-side implementation for providing the content, since everything could be done seperately, using the `ajax_controller`. Furthermore the `dashboard.js` script generates buttons to update the data currently displayed, which essentially is just another ajax request to the Twimight Server.

A considerable amount of time has been invested in design of the web interface. Since the web interface is accessed by users and not by machines, as it is the case for the API, graphical design becomes important. We basically relied on *cascading stylesheets* for the whole design of the different elements of the page. A noteworthy point is the inclusion of custom fonts, which are also provided by the Twimight Server. This greatly benefits the uniqueness of the website. Another feature, worth noting, is the clustering within the Google Map. Since a Twimight client can store many location reports on the server, the map would generally be overcrowded and therefore the clustering was added to improve the map. As previously mentioned, the resulting web interface is shown in fig. 3.1.

Figure 4.3: Error message on the web interface after an invalid request.

Programmer - an organism that turns coffee into software.

Unknown

Chapter 5
Conclusion

In this final chapter we conclude this thesis by first sharing some thoughts on the thesis, including personal thoughts on what has been learned, and second by looking at possible future work that is closely related to the Twimight Server in section 5.1.

Looking back at the thesis as such, we notice the many things that have been learned. The scope of technologies, standards and environments that were involved was particularly broad and even if the task sounded simple at first, it yet provided many challenges, which had to be solved.

We had to evaluate how to deal with all the technologies, software, environments and standards involved and how to acquire the relevant knowledge. Therefore we decided that it is to be preferred to rather read a book on a specific topic, than to just google for the answer. This definitely makes knowledge on this topic much more persistent, and certainly results in a much better overall solution to a given problem.

After making this decision, we acquired knowledge on various topics, as for example OpenSSL, certificates and the processes a certificate authority has to deal with; further we greatly deepened our abilities in modeling a problem and the use of design patterns; and most notably we were able to improve our programming skills in PHP and MySQL.

While developing not only technical problems arose, but also aesthetical and design specific questions had to be tackled. We had to decide on how to model the problem, i.e., to which level of granularity we separate the data structures, and when to stop separating. The result of this modeling process can be analyzed in chapter 4.

5.1 Future Work

Despite the fact that the current implementation of the Twimight Server is working pretty much flawlessly, there are several open issues and numerous ideas, which might be approached in the future. Below a list is provided, in order to sketch the work for the future.

Integrate the Twimight Client Application

The first and most important future task is to implement connection to the Twimight Server API in the Twimight client application, since without that, the Twimight Server does not make any sense.

Scalability Considerations

Having the ever-growing Twitter user community in mind, the question of scalability with respect to the Twimight Server is quite a legitimate one. The current implementation most certainly is not perfectly efficient and, hence, needs to be reviewed and improved accordingly.

Advanced Neighbor Finding

Currently, as emerged from chapter 4 above, neighbor finding is done by considering the most recent location reports of other users and the requesting user's current location report. This basic functionality might in the future be extended as to provide heuristics for neighbor finding. As an example one might think of selecting neighbors based on locations visited frequently, on location followers and not only on the current location of the requesting user.

MySQL-CA only

As already stated above, OpenSSL's certificate authority functionality is based on files and folders, whereas we wanted to base the Twimight Server's CA functionality in a MySQL database. Consequently, many things are done twice, which clearly is an overhead. To further increase the performance of the Twimight Server the implementation of a MySQL only CA is desirable. This can be done by still using OpenSSL, but rather than its commandline interface, we could make use of the library OpenSSL provides, to write small C programs.

Bullet-proof the Server

Although we considered attack vectors like cross-site scripting (XSS), cross-site request forgery (XSRF or CSRF) and SQL injection (SQLi), the risk of vulnerabilities remains. Therefore it is reasonable to perform detailed security audits to possibly find and then remove vulnerabilites.

Publish Tweets

A very useful, and quite easy implementable feature would be to enable the Twimight Server to push tweets to Twitter, hence, acting as a relay. Imagine the following case during a disaster:

Client A, B and C can communicate using Twimight and its delay tolerant Bluetooth network. While tweets are being sent by A, B and C, client C is all of a sudden able to connect to the fixed-infrastructure network.

The *Publish Tweets* feature would now enable client C to push all *signed* tweets, including those by A and B, to the Twimight Server, who in eturn pushes these tweets to Twitter. Having this feature implemented is definitely of great use and may, given the above circumstances in a disaster case, contribute to disaster awareness.

Disaster Awareness and Sensor Data

To further contribute to disaster awareness, the Twimight Server could also process *sensor data tweets*, which were sent by Twimight clients, and produce a mapping of the disaster in spatial, as well as temporal manner.

Arbitrary Social Network Integration

Twimight, obviously, provides means of tweeting during disasters. However, communication during disasters should not require an affected person to use Twitter, but rather allow for arbitrary social networks to work in conjunction with Twimight. The Twimight Server would be the entity that is able to connect to all these social networks and provide its API to clients to take full advantage of it for disaster relief. Social network APIs should only be required to support OAuth, which many in fact do. Despite this usefulness, implementing this functionality requires many changes in the current implementation and is a very demanding task.

Make the Twimight Server API Public

In disasters, ideally, arbitrary clients can exchange messages, provided they have some kind of communication interface. Even though Twimight and its server provide means of communication for disaster events, they have to provide further services. In order to fully use the Twimight Servers capabilities, the server needs to provide an application programming interface (API), similiar to the Twitter API, where one can register applications to exchange data with the Twimight Server.

Teaching Twimight Server Privacy

Up to this point, all data, which is sent to the Twimight Server by clients, is stored on the server and might potentially be used for answering requests by

an arbitrary other client. As an example one might consider, that client A has stored his MAC address, as well as some location reports on the server. Under appropriate conditions the MAC address of client A will be sent to client B, upon a request for neighboring users by client B. In this process client A has only very binary control over his data: He can either enable or disable Twimight Server communication.

A potential future task may, hence, be the implementation of different levels of privacy. The respective level may, for example, be changed using the web interface. A user might then adjust this privacy level at will. Nevertheless, enabling this privacy options might result in serious problems regarding opportunistic communication. As an example one might imagine a disaster case with a set of users close to each other, but each with an elevated privacy level. This could potentially result in disabling communication even though very well possible and probably needed as well. Consequently, the potential drawbacks have to be evaluated and analyzed, in order to best enable the user to protect his privacy, while still maintaining and guaranteeing Twimight core functionality.

Improve the Web Interface

- **i18n**
  The web interface is currently only available in english. However it would be much more convenient to have the possibility of serving the web interface in arbitrary languages. This process is known as *internationalization* or, for short, *i18n*. Providing i18n is closely related, or, more precisely, can be achieved easily by extending the web interface by a templating engine, as described just below.

- **Template Engine**
  Using a template engine, such as for example *Smarty*[1] can be of great use for a web interface. Deploying a template engine enables the developer to focus on the implementation of the actual "high-level" business logic, rather than having to focus on low-level problems. At the moment, the Twimight Server implementation does not use a template engine, since the amount of template files is not too large yet, or, in other words, the template files are still manageable. However with increasing complexity of the tool, it definitely makes sense to deploy a template engine.

- **Session Management**
  PHP offers many ways of handling sessions. In the current implementation of the Twimight Server, all session related information is stored in files in the directory `data/sessions/`, relative to the Twimight Server root directory. A future implementation of the server could move all session related information, i.e., the *session identifier* and session variables, to

[1] `http://www.smarty.net`

the MySQL database, where rows might even be linked to the user table, thereby offering new analysis and log possibilites.

Resolve Known Bugs

Upon reviewing the code, and thanks to the *Network Security* lecture, the following bugs, or more accurately, set of bugs, have been disclosed:

- **Cross-site Request Forgery**
  Currently, there is a CSRF issue for users logged in to the web interface. If malicious websites issue a request directly to either

  ```
  [twimight-domain]/ajax/certificate/revoke
  ```

  or to

  ```
  [twimight-domain]/dashboard/user/delete
  ```

  those actions are executed without further human interaction, with the obvious consequences for the user currently logged in. A possible solution to prevent this attack, is to add for example Google reCAPTCHA. Since the two actions prone to the attack are not executed on a daily basis, the overhead for the user remains acceptable.

- **Cross-browser Compatibility**
  Despite the beauty of the web interface in modern browsers, there are most likely quite numerous design issues in older browsers and, needless to say, in Internet Explorer. Those issues need to be approached and solved.

In conclusion we notice that the foundation of the Twimight Server has been laid and various tasks for future work exist.

Appendix A

Twimight Server Setup

1. Make sure that you have `Ubuntu 11.10`, as well as the following packages installed on your system:
   - `lighttpd`
   - `openssl`
   - `mysql-server`, `mysql-client`, `mysql-common`
   - `php5`, `php5-curl`, `php5-mysql`, `php5-cgi`, `php5-cli`
2. Checkout a copy of the source code from the Google code repository to a temporary folder, say `/tmp/twimight_server`.

   ```
   svn checkout \
   http://twimight-disaster-server.googlecode.com/svn/trunk/ \
   /tmp/twimight_server
   ```

3. Export your working copy to the destination folder, where you want the Twimight Server to reside, e.g. `/srv/twimight_server`.

   ```
   svn export /tmp/twimight_server /srv/twimight_server
   ```

4. Change to the `script/install` folder, execute the install script and provide the necessary information:

   ```
   cd /srv/twimight_server/script/install
   ./install.sh
   ```

Appendix B

Twimight Server Development

In this appendix, we provide some hints and explanations that might be useful when continiuing Twimight Server development.

Code Repository

For version control purposes we created a Google code repository, to be found under `http://code.google.com/p/twimight-disaster-server`. We used a *Subversion* repository. Information on Subversion can be found for example in the following book: `http://svnbook.red-bean.com`.

Editor

To edit the code we relies on the powerful *Emacs* editor, with the PHP mode, which is available under `http://php-mode.sourceforge.net`. The PHP mode was used for automated intendation and code highlighting.

Clean Code

Whether to include comments in the code or not is an ongoing and highly controversial discussion. For this project we decided *to not comment* any source code, but rather to give variables and functions a very meaningful, self-explanatory name. From our point of view, this approach is as good as comments.

Appendix C

JavaScript Object Notation

Below we specify the JavaScript Object Notation using the diagrams provided at `http://json.org`. These diagrams completely specify JSON, except minor encoding details. For the MIME type `application/json` please refer to [6].

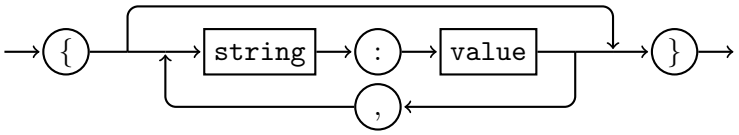


Figure C.1: JSON `object`

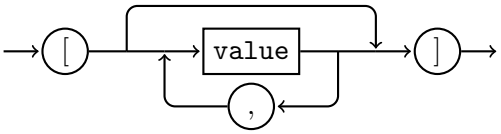


Figure C.2: JSON `array`

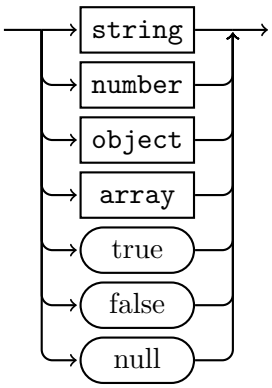


Figure C.3: JSON `value`

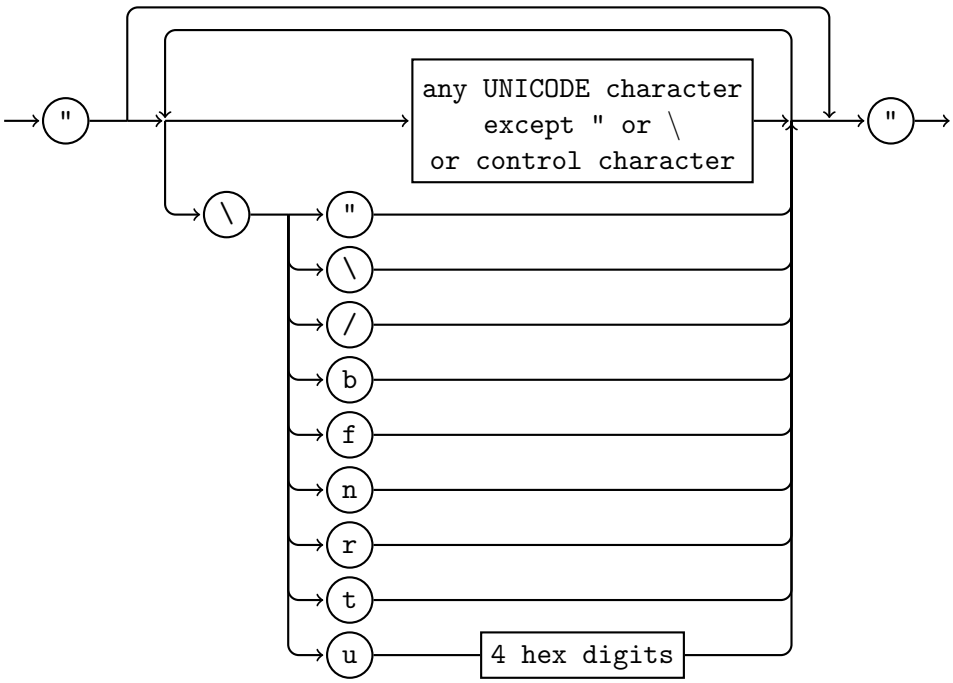


Figure C.4: JSON `string`

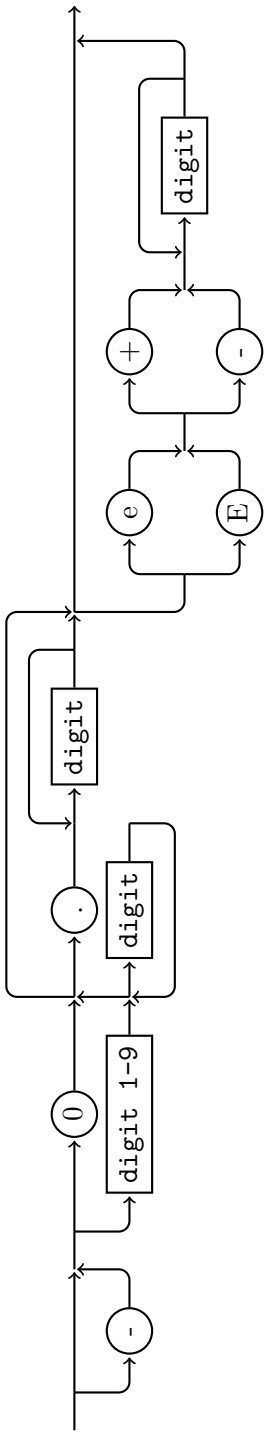


Figure C.5: JSON `number`

Appendix D

Database Structure

Table D.1: Structure of table server_credentials

Column	Type	Null	Default
id	int(10)	No	
consumer_key	char(22)	No	
consumer_secret	char(43)	No	

Table D.2: Structure of table user

Column	Type	Null	Default
id	int(10)	No	
first_time_seen	timestamp	No	CURRENT_TIMESTAMP

Table D.3: Structure of table user_certificate

Column	Type	Null	Default	Links to
id	int(10)	No		
user_id	int(10)	No		user (id)
serial_number	int(10)	No		
issued	timestamp	No	CURRENT_TIMESTAMP	
valid_from	timestamp	No	0000-00-00 00:00:00	
valid_to	timestamp	No	0000-00-00 00:00:00	
revoked	timestamp	Yes	NULL	
certificate	text	No		

Table D.4: Structure of table user_connection

Column	Type	Null	Default	Links to
user_id	int(10)	No		user (id)
type	enum('bluetooth', 'wifi')	No		
mac	varchar(17)	No		
lts	timestamp	No	CURRENT_TIMESTAMP	

Table D.5: Structure of table user_credentials

Column	Type	Null	Default	Links to
id	int(10)	No		
user_id	int(10)	No		user (id)
server_credentials_id	int(10)	No		server_credentials (id)
access_token	varchar(100)	No		
access_token_secret	varchar(100)	No		
revoked	datetime	Yes	NULL	
expiry	datetime	Yes	NULL	

Table D.6: Structure of table user_follower

Column	Type	Null	Default	Links to
user_id	int(10)	No		user (id)
follower_id	int(10)	No		user (id)

Table D.7: Structure of table user_location_report

Column	Type	Null	Default	Links to
user_id	int(10)	No		user (id)
latitude	double	No		
longitude	double	No		
accurracy	int(10)	No		
timestamp	datetime	No		
provider	enum('gps', 'network')	No		

Appendix E

References

[1] Aljohani, Naif; Alahmari, Saad and Aseere, Ali. *An organized collaborative work using twitter in flood disaster*. In *ACM Web Science 2011*. March 2011.

[2] Beaulieu, Alan. *Einführung in SQL*. O'Reilly, 2nd edition, August 2009. ISBN 978-3-89721-937-3.

[3] Bogus, Andre. *Lighttpd*. Packt Publishing, October 2008. ISBN 978-1-84719-210-3.

[4] Carta, Paolo. *Implementation of a disaster mode to maintain twitter communications in times of network outages*, August 2011.

[5] Chowdhury, Abdur. *Global pulse*, 2011. URL `http://blog.twitter.com/2011/06/global-pulse.html`.

[6] Crockford, D. *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627 (Informational), July 2006. URL `http://www.ietf.org/rfc/rfc4627.txt`.

[7] Dynes, Russell Rowe. *Organized behavior in disaster*. Disaster Research Center, University of Delaware, 1985.

[8] Fielding, R.; Gettys, J.; Mogul, J.; Frystyk, H.; Masinter, L.; Leach, P. and Berners-Lee, T. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Draft Standard), June 1999. URL `http://www.ietf.org/rfc/rfc2616.txt`. Updated by RFCs 2817, 5785, 6266.

[9] Framework, Zend. *Recommended project directory structure*. URL `http://framework.zend.com/manual/en/project-structure.project.html`.

[10] Gamma, Erich; Helm, Richard; Johnson, Ralph and Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, November 1994. ISBN 978-0-201-63361-0.

[11] Hammer-Lahav, E. *The OAuth 1.0 Protocol*. RFC 5849 (Informational), April 2010. URL `http://www.ietf.org/rfc/rfc5849.txt`.

[12] Hossmann, Theus; Carta, Paolo; Schatzmann, Dominik; Legendre, Franck; Gunningberg, Per and Rohner, Christian. *Twitter in disaster mode: Security architecture*. In *CoNext Special Workshop on the Internet and Disasters*. ACM, December 2011.

[13] Hossmann, Theus; Legendre, Franck; Carta, Paolo; Gunningberg, Per and Rohner, Christian. *Twitter in disaster mode: Opportunistic communication and distribution of sensor data in emergencies*. In *ExtremeCom 2011 - The Amazon Expedition*. 2011.

[14] Hughes, Amanda Lee and Palen, Leysia. *Twitter adoption and use in mass convergence and emergency events*. In *Proceedings of the 2009 ISCRAM Conference*. 2009.

[15] Jain, Sushant; Fall, Kevin and Patra, Rabin. *Routing in a delay tolerant network*. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '04, pages 145–158. ACM, 2004. ISBN 1-58113-862-8.

[16] JESS3. *The geosocial universe 2.0*. URL `http://jess3.com/geosocial-universe-2/`.

[17] Krieg, Michael. *Lighttpd - kurz & gut*. O'Reilly, 1st edition, April 2009. ISBN 978-3-89721-549-8.

[18] Mednieks, Zigurd; Dornin, Laird; Meike, G. Blake and Nakamura, Masumi. *Programming Android*. O'Reilly Media, 1st edition, August 2011. ISBN 978-1-4493-8969-7.

[19] Mendoza, Marcelo; Poblete, Barbara and Castillo, Carlos. *Twitter under crisis: can we trust what we rt?* In *Proceedings of the First Workshop on Social Media Analytics*, SOMA '10, pages 71–79. ACM, 2010. ISBN 978-1-4503-0217-3.

[20] Mills, Alexander; Chen, Rui; Lee, JinKyu and Rao, H. Raghav. *Web 2.0 emergency applications: How useful can twitter be for emergency response? Journal of Information Privacy & Security*, volume 5(3), 2009.

[21] Nystrom, M. and Kaliski, B. *PKCS #10: Certification Request Syntax Specification Version 1.7*. RFC 2986 (Informational), November 2000. URL `http://www.ietf.org/rfc/rfc2986.txt`. Updated by RFC 5967.

[22] OpenSSL. *Openssl project documents*. URL `http://openssl.org/docs`.

[23] Reenskaug, Trygve. *Mvc*. URL `http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html`.

[24] Rescorla, E. *HTTP Over TLS*. RFC 2818 (Informational), May 2000. URL `http://www.ietf.org/rfc/rfc2818.txt`. Updated by RFC 5785.

[25] Shklovski, Irina; Palen, Leysia and Sutton, Jeannette. *Finding community through information and communication technology in disaster response.* In *Proceedings of the 2008 ACM conference on Computer supported cooperative work*, CSCW '08, pages 127–136. ACM, 2008. ISBN 978-1-60558-007-4.

[26] Viega, John; Messier, Matt and Chandra, Pravir. *Network Security with OpenSSL.* O'Reilly Media, 1st edition, June 2002. ISBN 978-0-596-00270-1.

[27] Williams, Abraham. *Twitteroauth php library.* URL `https://github.com/abraham/twitteroauth`.