

# Girji

## *Metacode extensibility in Girji*

---

**Simen Lomås Johannessen**

INF-3981 - Master's Thesis in Computer Science, June 2014



# Abstract

Online web services that store fitness and health related data is evolving to provide useful services to end-users. Examples of this include RunKeeper, Fitbit and MS HealthVault. These services interconnect creating an ecosystem of online web services. The services they provide are mainly targeted for the consumer market. However, professional sport clubs may potentially benefit from integrating these systems in their daily activities. This, however, requires a different set of data analytic than commonly provided by these services.

A problem is connecting this type of ecosystems with statistical analytic tools like R, Matlab, and Excel for doing statistical analytics and machine learning. There is privacy concerns related to sensitive data and misuse. This thesis explores this problem creating an extensible system for doing analytic with statistical analytic tools on online data archives.

The system integrates the web services Fitbit and RunKeeper and creates a runtime support for analytics written in R and Python. The system encapsulates the burdens of privacy concerns and authentication for interacting with web services. The system implements operational consent to give athletes high level of control how data is used in analytics over long periods of time. This is provided through metacode abstraction and eligibility checking. Metacode extends the runtime dynamically by being able to run user provided code that for instance can check for privacy violations upon data accesses. The evaluation showed that Fitbit and RunKeeper as data archives for analytics have some constraints in concern of latency and rate limits. With caching and preemptive crawling the web services can become useful data sources for professional sport clubs to integrate with.



# Acknowledgements

I would like to thank my supervisor Professor Dag Johansen and Dr. Håvard Dagenborg Johansen for valuable input and useful critiques of this research work, during the development of this thesis.

I want to thank my family for support during my years with studying.

Further, I would like to express my appreciation to the best friend you can get, Ida Jaklin "Energizer" Johansen. Thank you for discussions and motivating me through this thesis and invaluable friendship over these past 5 years.

Lastly, I want to thank my friends Simon Engstrøm Nistad, Alexander Svendsen, Tom Pedersen and rest of the class for the great companionship in the past 5 years.



# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.2	Problem definition . . . . .	2
1.3	Method . . . . .	3
1.4	Interpretation . . . . .	3
1.5	Context . . . . .	4
1.6	Terminology . . . . .	5
1.7	Outline . . . . .	5
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	Background . . . . .	7
2.1.1	Technical background . . . . .	7
2.1.2	Big Data and machine learning . . . . .	7
2.1.3	JSON . . . . .	8
2.1.4	MongoDB . . . . .	9
2.2	OAuth . . . . .	9
2.3	Data sources and Web Services . . . . .	10
2.3.1	RunKeeper . . . . .	10
2.3.2	Fitbit . . . . .	11
2.3.3	ZXY Sport Tracking System . . . . .	12
2.4	Remote Procedure Call (RPC) . . . . .	13
2.5	Access Control List . . . . .	15
2.6	Broker design . . . . .	15
2.7	Related work . . . . .	16
2.7.1	MapReduce . . . . .	16
2.7.2	Mobile Code . . . . .	16
2.8	Summary . . . . .	17
<b>3</b>	<b>Requirement Specification</b>	<b>19</b>
3.1	Functional Requirements . . . . .	19

3.1.1	Introduction . . . . .	19
3.1.2	Overview . . . . .	20
3.2	System Model . . . . .	21
3.2.1	Overview . . . . .	21
3.2.2	Client . . . . .	21
3.2.3	Back-end . . . . .	22
3.2.3.1	Process . . . . .	22
3.2.4	Runtime . . . . .	22
3.2.4.1	API . . . . .	22
3.2.4.2	Data acquisition . . . . .	22
3.2.4.3	Access Control . . . . .	23
3.2.5	Storage . . . . .	23
3.2.6	Interface . . . . .	23
3.3	Non-Functional Requirements . . . . .	24
3.3.1	Extensibility . . . . .	24
3.3.2	Interoperability . . . . .	25
3.3.3	Dependency and Availability . . . . .	25
3.3.4	Security and Privacy . . . . .	26
3.3.5	Performance . . . . .	26
3.3.6	Scalability . . . . .	27
3.3.7	Accessibility . . . . .	27
3.3.8	Usability . . . . .	27
3.4	Summary . . . . .	28
<b>4</b>	<b>Design</b>	<b>29</b>
4.1	System architecture . . . . .	29
4.1.1	System models . . . . .	29
4.1.2	Overview . . . . .	32
4.2	Back-end . . . . .	33
4.2.1	Overview . . . . .	33
4.2.2	Metacode . . . . .	34
4.2.3	Cache . . . . .	35
4.2.4	Anonymizer . . . . .	36
4.3	Language library . . . . .	36
4.4	Access Control List and Projects . . . . .	37
4.5	Client . . . . .	38
4.6	Authentication . . . . .	39
4.7	Storage . . . . .	39
<b>5</b>	<b>Implementation</b>	<b>41</b>
5.1	R-library . . . . .	41



5.1.1	Initiating . . . . .	41
5.1.2	Remote Procedure Call (RPC) . . . . .	42
5.1.3	R Cache . . . . .	43
5.2	Runtime and WebServer . . . . .	44
5.2.1	Running data processing functions . . . . .	45
5.2.2	Metacode . . . . .	46
5.2.3	Cache . . . . .	47
5.2.4	Anonymizer . . . . .	48
5.3	Client . . . . .	48
5.4	Database . . . . .	49
5.5	Webpage, authentication and users . . . . .	49
<b>6</b>	<b>Use-Case</b>	<b>53</b>
6.1	Analytic functions . . . . .	53
6.1.1	Sleep and Steps . . . . .	53
6.1.1.1	Result . . . . .	56
6.1.2	Steps . . . . .	56
6.1.3	Running . . . . .	56
6.1.4	Sleep cycle . . . . .	57
6.2	Web site . . . . .	58
6.2.1	Services . . . . .	58
6.2.2	Access Control List . . . . .	59
6.2.3	Metacode . . . . .	60
6.2.4	Project . . . . .	60
<b>7</b>	<b>Evaluation and Results</b>	<b>61</b>
7.1	Methods . . . . .	61
7.2	Experiments and Results . . . . .	62
7.2.1	Experiments Setups . . . . .	62
7.2.2	Experiments . . . . .	63
7.2.2.1	Experiment 1 . . . . .	63
7.2.2.2	Experiment 2 . . . . .	65
7.3	General Discussion . . . . .	66
7.3.1	Extensibility and Interoperability . . . . .	66
7.3.2	Security . . . . .	67
7.3.3	Privacy concerns . . . . .	68
7.3.4	Accessibility . . . . .	69
7.3.5	Scalability . . . . .	70
7.3.6	Fault Tolerance . . . . .	70
7.3.7	General experiences . . . . .	71

<b>8</b>	<b>Concluding Remarks</b>	<b>73</b>
8.1	Achievements . . . . .	73
8.2	Conclusion . . . . .	74
8.3	Future Work . . . . .	75
	<b>References</b>	<b>75</b>

# List of Acronyms

<b>TIL</b>	Tromsø Idrettslag
<b>JSON</b>	JavaScript Object Notation
<b>HTML</b>	HyperText Markup Language
<b>CSS</b>	Cascaading Style Sheets
<b>API</b>	Application Programming Interface
<b>HTTP</b>	Hypertext Transfer Protocol
<b>URL</b>	Uniform Resource Locator
<b>HMAC</b>	Hash-based Message Authentication Code
<b>ACL</b>	Access Control List
<b>LRU</b>	Least Recently Used
<b>URL</b>	Uniform Resource Locator
<b>RPC</b>	Remote Procedure Call
<b>REV</b>	Remote Evaluation
<b>IOT</b>	Internet of Things
<b>ZXY</b>	ZXY Sport Tracking System
<b>OS</b>	Operating System
<b>STDOUT</b>	Standard Output
<b>STDIN</b>	Standard Input
<b>STDERR</b>	Standard Error
<b>IDE</b>	Integrated Development Environment

**IPC** Inter-process Communication

**RDBMS** Relational Database Mangement System

**PDF** Portable Document Format

# List of Figures

2.1	The dashboard presenting various statistics. A activity from Fitbit is also integrated in the fitness feed . . . . .	11
2.2	Screenshot from Fitbit.com showing several tracking devices currently on the marked. Fitbit Flex is located left in the picture	12
2.3	Overview of the ZXY system with receivers placed around the pitch, and players wearing sensors. Image from zxy.no . . . . .	13
2.4	A simplified illustration of Java RMI. . . . .	14
3.1	Abstract Architecture . . . . .	21
4.1	a) Client-WebService architecture. . . . .	30
4.2	b) Client-Proxy-WebService architecture. . . . .	30
4.3	System architectural overview . . . . .	31
4.4	Functions chained where the output are sent as input to the next function . . . . .	33
4.5	The figure illustrates how metacode is inserted into the loop when an analytic function does a method invocation. If there is any errors while running the metacode this is forwarded to the executing data analytic function following the flow 6-7-8. .	34
4.6	The language library interacts with the runtime to request data. In addition, the library can use the file system to store output from the analytic function. . . . .	37
4.7	Data analytic functions are signed before sent to the runtime for execution . . . . .	38
5.1	GUI application. The file listed first is the first to be executed	48
5.2	HTML of the Access Control List page. Curly brackets define input of dynamic data. In this page a list of ACLs is inserted, where each object contains a resource and permission key-value object . . . . .	51

6.1	Plot of number of steps taken and the number of times restless during the following night sleep. . . . .	54
6.2	Plot of number of steps taken and the amount of time in minutes restless during the following night sleep. . . . .	54
6.3	Plot of number of steps taken and the efficiency of the sleep the following night sleep. Efficiency is a percent calculated by Fitbit . . . . .	55
6.4	Plot of number of steps taken and the number of times awakened during the following night sleep. . . . .	55
6.5	Daily step count over a range of dates. The date is hidden for privacy reasons. . . . .	56
6.6	Plot of time in seconds against the number of meter done during a run. . . . .	57
6.7	Plots the stages of sleep during the night. In the x-axis the time asleep in minutes. In the y-axis the different stages: Awake, Rapid Eye Movement (REM), deep sleep. . . . .	57
6.8	User sign up page. User fills the sign up form to create a user	58
6.9	Services page. All web services available are listed here . . . .	59
6.10	Access Control List page. To create new resource the user enters the resource and sets what access permission it should have. . . . .	59
6.11	Metacode . . . . .	60
6.12	Simple project registration. Only three resources is listed for demonstration purposes . . . . .	60
7.1	Latency in ms when multiple clients runs experiment 1 in parallel	65

# List of Tables

4.1	Database Schema . . . . .	40
5.1	API . . . . .	44





# Chapter 1

## Introduction

### 1.1 Background

A growing number of people are focusing more on fitness and exercising more [2]. That is reflected in the increasing amount of health applications and tools for self-tracking on the market. The term *quantified self* is used to describe this movement, described as gaining knowledge about yourselves through numbers [37]. This movement is also reflected in the increasing number of devices that is connected to the Internet every day [42] creating a Internet of Things (IOT). Wearables for self-tracking that is connected to the Internet create an ecosystem for everything related to fitness and health. They are rapidly improving, providing better accuracy and becoming affordable for consumers. Information they can measure ranges from daily number of steps, distance covered, calories burnt and sleep quality. Added on top of the expanding market for wearables, there is also a growing market for health applications for smart phones [46]. Smart phones can provide much of the same information begin equipped with accelerometer and GPS tracking.

Many companies that collect and store health related data have their data silos open for third parties to integrate with. Examples of this include RunKeeper [18], Fitbit [17] and MS HealthVault [3]. This can benefit both consumers and developers wanting to create own services and applications based on the data the companies collect and store. For instance, Fitbit, offers an Application Programming Interface (API) for developers to integrate to gain access to users data. All the information Fitbit devices track is uploaded to their data silos and is then made available to view at a personal web

portal.

Sports analytic are a increasingly hot topic [11]. Tromsø Idrettslag (TIL), a Norwegian soccer club, uses ZXY Sport tracking<sup>1</sup> to collect data while playing matches and training. The ZXY system intended for professional use, provides a higher data granularity than the typically consumer products. Its use range is limited to the installation location. Other sport analytic systems include the arena sports analytics system Bagadus [41] building on ZXY data and the notational analysis system Muithu [21]. For professional sport clubs, integrating and using wearables in addition to the professional equipment, the blind spots during the rest of the day can be tracked, and potentially gaining new insight regarding players fitness and health.

Using analytic on the data collected from wearables and health applications can help to potentially learn more about a persons fitness and health. For instance, when something abnormal is detected, it can be dealt with early and long injuries are prevented, possibly extending the sport carrier for athletes [16] [47].

## 1.2 Problem definition

A large ecosystem of interconnected online web services that store, analyse, and visualize personal data is evolving to provide useful services to end-users. Although primarily targeted for the consumer marked, professional sport clubs may potentially benefit from integrating these systems in their daily activities. This, however, requires a different set of data analytic than commonly provided by these services.

*This thesis will explore the problem of connecting statistical analytic tools like R, Matlab and Excel to personal body sensor data archives like FitBit, RunKeeper, Python and the ZXY Sport tracking system. A language binding and runtime will need to be constructed. The runtime will be dynamically extensible by user-code to give users high control of what data that is accessible. Possibility for privacy preserving will be taken into concern in the system design. Several analytic functions will be developed to demonstrate the system and end-to-end latencies will be measured*

---

<sup>1</sup><http://www.zxy.no/>

## 1.3 Method

The final report of the ACM Task Force on the Core of Computer Science [8] divides the discipline of computing into three major paradigms:

- *Theory*: Theory: Rooted in mathematics, the approach is to define problems, propose theorems and explore to prove them in order to find new relationships and progress in computing.
- *Abstraction*: Rooted in the experimental scientific method, the approach is to analyze a phenomenon by creating hypothesis, constructing models and simulations, and analyzing the results.
- *Design*: Rooted in engineering, the approach is to state requirements and specifications; design and implement systems that solve the problem, and test the systems in order to find the best solution to the given problem.

For this thesis, the design process seems to be the most suitable out of the three paradigms. The design process consists of 4 steps, which are repeated if tests reveal that the latest version of the system does not meet the requirements.

- *State requirements and specification*: A need or problem is identified, researched, and defined.
- *Design and implement the system*: Data models and a system architecture are designed. Prototypes are implemented.
- *Test the system*: Assessment and testing of prototypes.

## 1.4 Interpretation

This thesis will look into designing and implementing a system enabling R and Python to use online data archives as Fitbit and RunKeeper. Online data archives store consumer sensible data, therefore, the system by design will have users privacy in focus. A high degree of extensibility will be focused on, for both allowing more data sources and statistical tools to be added, and to provide dynamic extensibility of the runtime. The system will give users fine-grained control of what data research projects can access from the data archives. This, to give the user improved privacy control in long term where possibly new body-sensors is added providing higher data granularity, new

data mining algorithms is discovered and the research projects specifications is changed.

The thesis goal is to design and deploy a working prototype. The system will be evaluated by experiments demonstrating the system prototype by a proof of concept and performance will be measured. At-least two statistical packages and two data archives will be working in the prototype to prove flexibility and extensibility. The system will be developed following the design process methodology stated in 1.3.

## 1.5 Context

The iAD (information Access Disruption) Centre researches fundamental structures and concepts for large-scale information access applications<sup>2</sup>. iAD gets funds from the Research Council of Norway as a Center for Research-based Innovation (SFI). Microsoft in collaboration with Accenture, Cornell University, Dublin City University, BI Norwegian School of Management and the universities in Tromsø (UiT), Trondheim (NTNU) and Oslo (UiO) direct the research work.

A project developed by iAD is Bagadus [41]. It is an arena sport analytic system-tracking players integrated with the ZXY Sport Tracking System. It enables automatic playback of annotated events. The playback can follow group of players or individuals by using the exact position. Presentation of the video is done by stitching a panorama video or switching cameras. The system is deployed at Alfheim Stadium (TIL, Norway).

Another system is Muithu [21], developed to help to lower the barrier of annotating events. It lets you annotate sequences of a game with entities like player and comment from mobile devices. It encapsulates predefined events to speed up the annotation process. This information will then be time synchronized with the video feed and made available for search on entity's to get the corresponding video feed.

A system currently under developing is Girji [25], providing management of body-sensor data for sport analytic focusing on providing operative consent. It is a security-centered system giving users high control of what data that can be used. The work done in this thesis can be in cooperated into Girji.

---

<sup>2</sup><http://site.uit.no/iad/>

## 1.6 Terminology

- *Athlete*: Are the athletes practicing sports who agree to be participants in scientific research projects by being the source of information collecting.
- *Analytical Principal (AP)*: Are those who will write the statistical analytic functions that will be run.
- *Analytic functions*: Files/scripts that contains the code to be run.
- *Client*: Software that the analytic functions are sent from.

## 1.7 Outline

- *Chapter 2*: Presents background information and related work to the project.
- *Chapter 3*: Describes the requirement specification.
- *Chapter 4*: Describes the design and implementation of the system.
- *Chapter 5*: Gives a demonstration of the system.
- *Chapter 6*: Evaluates and discuss the system.
- *Chapter 7*: Concludes.



# Chapter 2

## Background

This sections presents some background related to the design and implementation.

### 2.1 Background

#### 2.1.1 Technical background

R is a programming language for statistical computing and graphics. Among statisticians and data miners R is widely taken into use [50] [13]. It is a interpreted language an the R source code is implemented in C, Fortran and R. For high computational heavy computations, code can be written in C, C++, Java and Python and called at runtime from R. Some of the features R provides are linear and nonlinear modeling, classical statistical tests, time-series analysis, classification and clustering. Strength of R is the user-produced packages, which provide much useful functionality for users.

Support for R as a statistical package was chosen for the positive reviews it as got among data miners [38] [35] [30].

#### 2.1.2 Big Data and machine learning

Big data is a term used to describe large quantities of data, both structured and un-structured. A large potential lies in examining and doing large scale

analytic on this kind of data; decision-making in businesses, health, and sports, helping to make more accurate analyses. The three Vs define big data: volume, velocity and variety [26]. Volume describes the large quantities of data it is. With Internet of Things (IOT) and wearable devices even more data will be generated. Variety of the data is about the data being represented in all kind of formats, both structured and un-structured. In addition, the data comes from various sources and data types like traditional databases, documents, web pages, and devices. Velocity describes the speed of the data both in and out.

In recent years two extras Vs have been added: value and veracity. Value refers to turn the data into money. A lot of data is fine, but if it cant be used to anything it is useless. Veracity refers to the quality and accuracy of the data. For instance, hash tags at Twitter, the reliability of tags is not high. High volumes come with a price in concern to quality and accuracy of the data.

Machine learning is a branch of artificial intelligence and linked up to big data. Tom M Mitchell defined machine as: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" [29]. The decision how to react is based on previous experience. New knowledge is learned if new measurements improve the performance.

Example of where machine learning can be useful includes fraud detection, product recommendation and churn prevention.

### 2.1.3 JSON

JSON<sup>1</sup> is a data interchange format. It has two structures for representing the data: collections of key value pairs and ordered list of values. Both are easily mapped to similar concepts in various programming languages. Collections of key value pairs are similar as dictionaries, struct, and hash tables in various other languages. Ordered list of values is a similar concept as arrays, lists and vectors. The two structures can be combined having a key value pair where the value is an ordered list of values. Handling data exchanging and mapping to languages are easy when the concepts matches data structures in programming languages.

---

<sup>1</sup><http://www.json.org/>



An example of a JSON data exchange is given below.

```
{
  "menu": {
    "id": "file",
    "value": "File",
    "popup": {
      "menuitem": [
        {"value": "New", "onclick": "CreateNewDoc()"},
        {"value": "Open", "onclick": "OpenDoc()"},
        {"value": "Close", "onclick": "CloseDoc()"}
      ]
    }
  }
}
```

### 2.1.4 MongoDB

MongoDB is an open-source document based database. Documents can be seen as a container for data, like a table in a Relational Database Management System (RDBMS). In a RDBMS there is rigid tables with static fields that will always be the same. MongoDB provides more flexibility in their schema design by being dynamic. Documents in a schema do not have to have the same field names or types. A document has a JSON-style representation called BSON, short for Binary JSON. It supports embedding documents in other documents. There can be other data types than JSON like Date type and BinData type. To provide high flexibility of aggregation functions MongoDB supports Map/Reduce data processing. It also has a rich query language.

## 2.2 OAuth

OAuth is an open standard for giving others access to an API in a secure, simple way. Many of the big players like Facebook, Google and Twitter uses OAuth. For service providers, OAuth, lets you open up your API to third parties, letting the they gain access to use data stored, without end-users spreading their password to their accounts across the web, protecting credentials. For consumer applications developers, OAuth lets you access the protected data on behalf of the user, and enrich your own applications

and services. There are mainly two protocol versions of OAuth in use, 1.0 and 2.0. Both requires the consumer application developer to register an application at the service providers website to get a consumer key and a consumer secret. Users have to authenticate the application, granting it permission. This is done at the service provider typically by being redirected to their website. Upon granting access, the user is redirected back to the consumer application with an access token. This access token is acting as the authentication of the user and has to be embedded in every request to the service provider. For OAuth 1.0 an access token and access token secret is returned in the callback. Requests to the service provider with OAuth 1.0 have to contain both the access token, and additionally the request have to be signed with the secret key. OAuth 1.0 is more complicated than version 2.0 and requires more work by the consumer developers.

OAuth can expose a security risk if the access token is lost or being eavesdropped. This can be dealt with limiting the access token lifetime and using SSL encryption. Any secret that cant be hidden doesnt count as a secret. The access token is confidential. Another threat is malicious applications pretending to be real ones, luring the consumers to grant access to them. More attacks are summarized in OAuth 2.0 Threat Model and Security Considerations [43].

## **2.3 Data sources and Web Services**

There are numerous web services in the domain of fitness and health. Two services that are integrated in our system are RunKeeper and Fitbit.

### **2.3.1 RunKeeper**

RunKeeper is a fitness and health ecosystem allowing you to store and retrieve body sensor data. Their core application is a running application for iOS, Android and Windows Phone tracking your exercises using movement-tracking features like GPS in mobile phones. All exercise is available through their website where health and fitness data is illustrated. Built around their success with the application they have developed a web portal for everything related to health and fitness. A screenshot of the web portal is shown below in figure 2.1. End-users can upload their own data that RunKeeper will store for them in the cloud taking care of storage costs, maintenance, backups and presentation of the data.

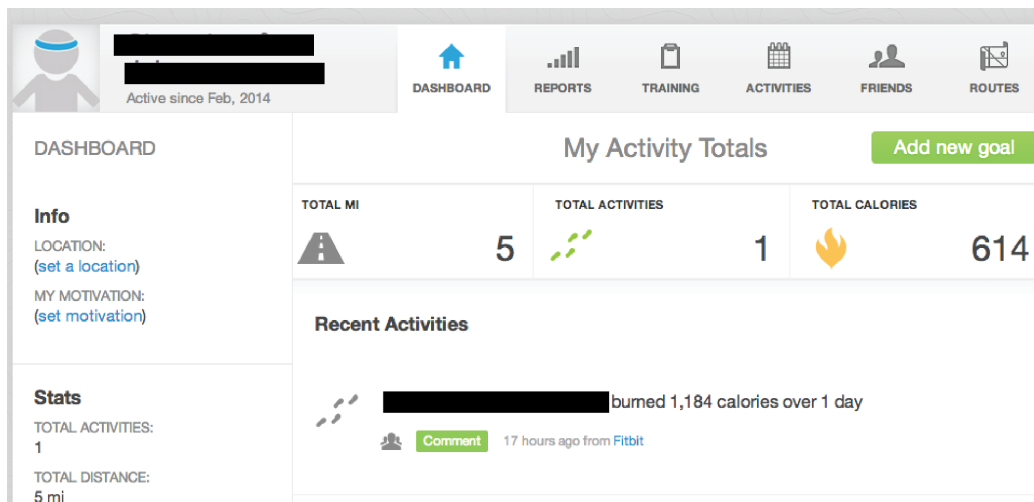


Figure 2.1: The dashboard presenting various statistics. A activity from Fitbit is also integrated in the fitness feed

In addition, they have introduced the HealthGraph API [18] where consumer developers can create own applications to take advantage of the data collected and stored. Data that can be fetched range from sleep, activities, body measurements, nutritions, and diabetes. Authentication to the Application Programming Interface (API) is done with the OAuth protocol. Healthgraph has a range of partners that they cooperate with enabling RunKeeper to provide an overview over all fitness tracking across devices and services. MyfitnessPal, Fitbit, WitThings and Zeo are some of their partners. The HealthGraph API can be reached over Hypertext Transfer Protocol (HTTP) with a RESTful [12] interface, supporting responses in JSON format. Being a RESTful interface, it takes advantage of the HTTP methods GET, POST, PUT and DELETE to fetch, add, modify or delete fitness activities and health measurements.

### 2.3.2 Fitbit

Fitbit has a range of devices where everyone integrates with their online web services by uploading data collected. This ranges from weight trackers to various activity trackers. The arm wristband, Fitbit Flex encourages you to be more active and generally live a healthier life. It tracks your daily activity by logging the number of steps you take, it tracks your sleep rhythm and duration with some manual effort. The technology inside is an accelerometer. By various algorithms it knows the difference between a hand movement and

a step. However, this is not 100 percent accurate, but good enough for most people providing a good estimate of daily activity. All the information is made available to browse through a personal web profile. Fitbit wires the devices only enabling them to upload data to Fitbit servers. The raw data cant be accessed.



Figure 2.2: Screenshot from Fitbit.com showing several tracking devices currently on the market. Fitbit Flex is located left in the picture

Fitbits API is split in two parts; a public API for everyone and a partner API. The partner API offers a higher granularity on the data. An example of this is how the public API only gives you the number of steps for a whole day, where the partner API provides you data down to steps per minute. To become a partner you have to send Fitbit an application explaining why your application should grant access. The applications are reviewed case by case.

Fitbit comes with a rate limit per hour per user. Currently, for the public API this is 150 requests per hour. Having a rate limit enables Fitbit to regulate the amount of requests to their service and thus provide a stable service by ensuring availability for everyone. The API is reached over HTTP with a RESTful interface, following the same principles as RunKeeper, but supports both JSON and XML data format in responses.

### 2.3.3 ZXY Sport Tracking System

ZXY Sport Tracking System (ZXY) is a professional sport analytic system that uses body sensors. Soccer teams in Norway, including Tromsø Idrettslag (TIL) and Rosenborg BK, use the system. Data captured is stored in a Sybase database with each match requiring about 500-700MB storage. The players have to wear a belt around their waist for the system to be able to track their movements. The ZXY system is able to track the players

movement very detailed with an accuracy of 0.5m. It has a resolution of 20 samples per second.



Figure 2.3: Overview of the ZXY system with receivers placed around the pitch, and players wearing sensors. Image from [zxy.no](http://zxy.no)

The technology behind it relies on a radio-based signaling substrate to provide real-time high-precision positional tracking, also including acceleration and heart rate [15]. An installation of receivers is required for the system to work. The home arena for TIL, Alfheim, is currently equipped with 10 receivers. A receiver tracks a specific area of the soccer field and combined they cover the whole pitch with some redundancy areas for fault tolerance. The communication from the belt to the receivers goes on a 2.45-5.2 GHz frequency radio signal. To compute the positional data the stationary radio receiver uses an advanced vector based processing of the received radio signal. The data is aggregated and stored into a relational database. Including the positions of the players ZXY also gives you the step frequency, speed and direction.

## 2.4 Remote Procedure Call (RPC)

Remote Procedure Call (RPC) is a form for inter-process communication. For the caller it is like using local code. In the background, the call is sent to

another process, with another address space, for executing. The procedure on the client will typically block until a response is received. A RPC should define how the data structure should be like including errors, requests and responses, in addition which serializer to use (JSON, XML, URI).

Java RMI [6] is one of the most known implementations of RPC. In short, shown by figure 2.4, it works by generating stubs and skeletons of the object interface to expose. When a client is invoking a remote method it makes use of the stub. The stub will marshal parameters and pass the request to the remote reference layer. The remote reference layer handles the lower level details of which invocation protocol to use on both the client and server side. At the server side the skeleton does an up-call to the remote object implementation, which does the actual method call. Parameters are serialized by marshalling and un-marshaling them when sent between the processes, handled by the stub and skeleton. The process is reversed on a response from the server. A cross-language implementation of RPC/RMI is Corba.

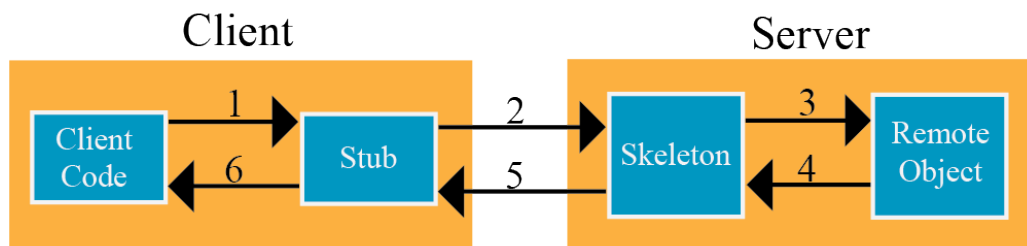


Figure 2.4: A simplified illustration of Java RMI.

Another approach for RPC is JSON-RPC [1] that encodes the procedural code in JSON. It is a lightweight approach to RPC simplifying distributed computing. Procedure calls and responses are done with a single JSON object. Parameters are wrapped in an array. The method is specified in an own key-value pair. The result response must contain an error key, set to null if there was no error. Any request has an ID attached to it to be able to match the request with the response message. This opens up for support of un-ordered responses. The protocol also supports notifications meaning a request doesnt require to have a response. JSON-RPC can be sent over any communication protocol like sockets, HTTP and pipes. An example of JSON-RPC request and response is given below.

```
{"method": "subtract", "params": [42, 23], "id": 1}
{"result": 19, "error" : null, "id": 1}
```

## 2.5 Access Control List

Access Control List (ACL) controls accesses to objects. An object has read and writes flags attached controlling if you can either read it, write it or both. The flags are individual for each user in the system. This means one user, Alice, can have read and write access, but another user, Bob, can only read the object. Orchestrating read and write accesses is typically implemented via a reference monitor that controls all accesses to objects [36], either denying or allowing the procedure to continue the operation.

In Unix ACL can be configured for files using the same principles as for objects. This includes read access, write access and execution access that can be set. This allows a user to execute the file. Accesses can be configured on a group level as well. For instance, it can be everyone with access to the machine, a set of users, or system administrators.

In the cloud, ACL can be used as authorization to map roles, principal or attributes of principals to set of rights on available services. A problem with ACL is it requires that the set of rights is predefined. With systems with many layers and services, ACL adds complexity with configuring ACLs across the system layers and services. A attempt to handle this is Code Capabilities (CodeCaps) [49]. Privileges are not predefined, but can evolve. Services do not require any access control list, preventing configuration of predefined access lists at every service/layer. Another similar approach of controlling authorization in cloud is Macarons [5].

## 2.6 Broker design

In the broker design pattern, the broker has responsibility for forwarding requests, exceptions and results, in a distributed setting, where components are decoupled [39]. Typically the components are heterogeneous. The broker will handle communication related concerns between the components. Benefits of a loose coupled system is in-dependability of other components where a component can develop, tested and maintained independently from the other components in the system. A potential issue is scalability with all having to go through one component.

## 2.7 Related work

This sections outlines descriptions of related work.

### 2.7.1 MapReduce

MapReduce [10] is a programming model typically used for doing computations on large data sets. MapReduce consists of two phases; the map procedure and reduce procedure. First a map function is run on input key-value pairs. The map phase will generate a set of intermediate key-value pairs; meaning similar keys are grouped together. This is where the reduce function comes into play. Reduce will run on each set of intermediate key-value pair generating a new key-value pair output. The computation is staged into separate parts. Multiple MapReduce jobs can be run after each other like in a pipeline, where each job works on the output of the previous in the pipe, creating a pipe of data processing jobs.

A basic example of a MapReduce job is a word counter job. The map phase receives a line of a text. It splits the line into words and generates a key-value pair of the word and a count number. This will sort all equal words together. The reduce phase will then run on the group of words counting occurrences. The output is a key-value pair with the word and a count of appearances in the input text.

Example of open source implementations of MapReduce is Cogset [48] and Hadoop [19].

### 2.7.2 Mobile Code

Mobile code is code sent over network between systems. Good examples here include JavaScript, Java applets and Flash, all popular choices for client side scripting. Mobile code is essentially code that can be executed on the client without explicit installing it.

Code mobility can be divided into two [4]: Weak mobility and Strong mobility. Strong moves the execution state (like stack and instruction pointer) of the code in addition to the code it-selves and data. This makes it possible to continue execute right where it left off. Weak only moves the code and the data.



Several paradigms exist in code mobility [4]:

- *Code on Demand (COD)*: code is requested when needed, and executed locally.
- *Remote evaluation (REV)*: code is sent over to a remote location where it is executed, and the result is returned.
- *Mobile Agent (MA)*: code migrates to another host, possibly with some intermediate results, and executed at the host.

COD and REV architectural styles can be categorized as weak mobility, and MA as strong mobility.

TACOMA [24] is an example of a system that deploys mobile agents from light clients, offloading computation to remote services. A similar approach is Agent Tcl [14]. Both systems were introduced in the mid 90s. An approach towards an agent-computing environment is TACOMA ACE [23]. It builds on previous work on TACOMA, creating an infrastructure for executing mobile agent code in a distributed setting. Another approach of software mobility is WAIFARER [22]. It is a desktop mobility approach where the desktop environment is moved around by saving and restoring application level state.

Some key benefits of mobile code involve a high degree of service customization. A problem with standard servers is their statically defined interfaces. When unforeseen client needs are discovered, programmers need to extend their interface to support the new functionality. Using a mobile code architectural approach, the client can solve the problem themselves, avoiding involving the service operators. It brings autonomy to the system.

## 2.8 Summary

In this chapter we have introduced background information for the thesis. We have taken a look at RunKeeper and Fitbit, and how to interact and take advantage of their services. In the last section we have looked upon some related work and concepts including: MapReduce and mobile code.



# Chapter 3

## Requirement Specification

This chapter outlines the requirement specification. Requirements and the proposed architectural design are influenced by work on Girji [25]. Functional and non-functional requirements are constructed with information provided in the previous chapter in mind.

### 3.1 Functional Requirements

#### 3.1.1 Introduction

A decision that influences the functional requirements is a wish of providing *operative consent*. In medical terms, consent is about getting permission for a treatment stating what is going to be done, including benefits and consequences of the procedure. Providing *operative consent* means in our context, before accessing data, the operation shall check if it is eligible to do it. This is to be able to secure and protect athletes privacy over a long time when there is possible changing environments like new body-sensors providing higher data granularity, new data mining algorithms and modifications in research projects specifications. In our experience, the willingness of participating in health researches lies on how good privacy is handled and if by participating directly benefits the athlete. In other words, if the direct outcome of the research can help athletes directly, the athletes are more willing to give away access to sensitive data.

There are two functional requirements that help to provide operative consent:

- *Dynamic extensibility at runtime*: Athletes shall have the possibility to control data access to their personal data through extensibility by code.
- *Eligibility*: Athletes shall have the possibility to set fine-grained control of what data that is accessible.

### 3.1.2 Overview

The envisioned system shall serve as a support for running analytics on data from online health and sport data archives like Fitbit and RunKeeper by using statistical tools. The system is envisioned used by APs in research projects on sports and health, and athletes to run analytics on their own data to learn more about health and sports at a professional level.

The system shall support uploading of user provided code, loading code dynamically and run it, thus extending the system functionalities. Extensibility through code is provided for athletes to dynamically control what data from the online archives that researches in projects are able to use by evaluating the data through coding.

The system shall give athletes fine-grained control of what data research projects can access. The athletes shall be able to control the exact data that analytic principals can use and access from data archives. This to give the athletes improved control and prevent losing control of their data sharing in changing environments. The system thus needs an interface for athletes to configure and set the access control lists.

The system will need to support graphical and textual output that is produced by the statistical analytic tools. The output is the end result after running an analytic function.

The system shall support chaining multiple analytic functions where the output from one analytic function can be sent in to the next one in the chain, staging the analytic process into separate parts. For instance, data can be filtered by the first function and used as the input of an aggregation function, summing up the result of the filtering process. This concept builds on passing of privileges. A scenario is where an athlete would like to share some of his data with an analytic principal to use, but the AP does not have privilege to extract the data directly. Thus the athlete can create a function that extracts the data required, and pass it to the AP. The AP can chain his own function with the athletes functions.

## 3.2 System Model

### 3.2.1 Overview

Based on the functional requirements we propose an abstract architecture of the system shown in figure 3.1. Components include a client and a back-end. The back-end consists of three components: process, runtime and storage. In addition, there is the data archive that the back-end interacts with.

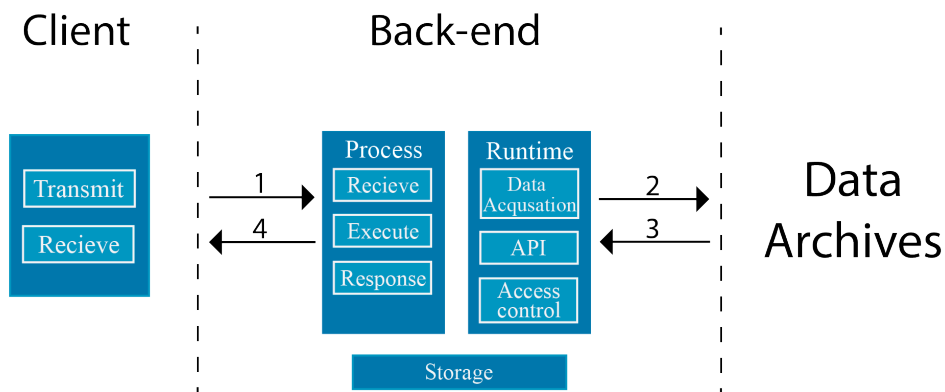


Figure 3.1: Abstract Architecture

The data flow is as following: the client interacts with the back-end process to request a execution of a analytic function, upon execution the analytic function interacts with the runtime to request various data, the runtime communicates with the online data archives. It can be multiple interactions between the analytic function and the runtime, and the runtime and the data archives, before a result is returned to the client.

### 3.2.2 Client

Client has to be able to *transmit* the data analytic function(s) to the runtime. The client application has to be able to specify the order of execution for the functions, if there is more than one. The client has to sign files for the back-end to be able to ensure files are authentic and not tampered with. The client has to be able to *receive* output generated by the execution. Upon a response from the back-end, the client has to store both graphical and textual output, if there are any.

### 3.2.3 Back-end

#### 3.2.3.1 Process

The back-end process has to be able to *receive* and persist files containing the analytic functions before executing them. The request authenticity has to be checked and aborted if not valid.

The back-end process has to be able to *execute* the analytic functions when receiving them. The back-end process has to orchestrate the whole operation by controlling input and output to the analytic functions in chains and alone. Many executions can be worked on simultaneously and the back-end process has to ensure there are no conflicts.

The back-end process has to be able to send a *response* to the client upon finished executions. Output generated from the last function executing has to be returned to the client. The back-end shall leave no traces of the execution after the client has received the result.

### 3.2.4 Runtime

#### 3.2.4.1 API

The runtime has to expose a way for the statistical analytic tools to interact with it to request data. The runtime has to examine the request and do appropriate actions out from the request.

#### 3.2.4.2 Data acquisition

The runtime must support *data acquisition* from the data archives. A high degree of extensibility is required to support new data archives like ZXY Sport Tracking System (ZXY) database. Data acquisition need to handle heterogeneous services that represent data differently and uses different API protocols. For instance, data responses and errors can be in different formats. In addition, online web services can have a diverse set of response times and availability. The data acquisition code needs to optimize data acquisition in concern of latency and availability.

### 3.2.4.3 Access Control

The runtime need to provide operative consent through *access control*. The data analytic function requests of data must be approved. The term *metacode* is used from work done in [20]. Techniques and concepts metacode utilizes include reference monitors, mobile code and extensible systems. Essentially, metacode is code that is attached to data segments. It can for example be run on every access to a specific data segment, providing *access control*. The metacode will extend the runtime dynamically providing extensibility in a dynamic manner. Extensibility of the runtime execution will be from a user perspective. Athletes will have the opportunity to upload metacode and thus configuring the runtime. Athletes will most likely not know how to configure and code the metacode. Therefore a set of pre-defined metacode can be deployed for the athlete to choose from. For expert users, self-uploading of code attaching the code to data will be supported. The back-end has to be able to store and persist metacode and relate it to data.

When a data access occurs a validation of eligibility must be done, either aborting or continuing the request. This resembles a reference monitor and will provide fine-grained control of what data that is accessible.

Summarized, operative consent is provided through access control. Athletes are ensured a high degree of customizability by letting them control how and what data that is accessible to APs.

### 3.2.5 Storage

The back-end has to have some form of storage to store credentials for athletes. Collecting personal data from Fitbit and RunKeeper requires OAuth authentication. This will generate access tokens and access token secrets. The tokens will be used in each request to the online data archives as authentication. In addition, upon a data access the storage will be asked about metacode to execute and eligibility of the request. Information about metacode and files has to be stored. The storage has to be a persistent storage that lives when the back-end crashes and is down.

### 3.2.6 Interface

To support operative consent the system needs to provide an interface. This is required for athletes to configure their data access control and upload

metacode.

## **3.3 Non-Functional Requirements**

### **3.3.1 Extensibility**

Extensibility is crucial for the system to allow it to evolve as new needs are discovered. New features need to be easy to add to the system without requiring a lot of effort and change in existing functionalities. A measure of high degree of extensibility is when new behavior of the system can be added without having to recompile the existing source code.

The system shall support extensibility to add new statistical tools, new web services and possible other data sources like databases being included in the system. There are various services and products out there. Whats the most used and highest ranked today may be forgotten tomorrow. Having to redesign the system when new data sources are added would not provide sustainability in long term. Analytic functions can be written in many statistical packages like R, Excel and Matlab. The system need to handle easy extension to new statistical tools as new needs is discovered. Supporting more statistical tools will lower the barrier for APs to do analytics letting them use a familiar tool.

In addition, the system shall support behavior extensibility to support metacode from athletes wanting to control data accesses done in the analytic functions. The system shall not have to be restarted to execute new metacodes received after start up of the system. However, the internal flow shall change when metacode is uploaded with the metacode being embedded into the loop, executed upon data accesses.

The communication between the statistical tools and the runtime has to be widely supported and simple to implement in order to support extensibility. The system extensibility shall be measured of how many statistical packages and online data archives that is supported, and if metacode is supported. At least the system shall be operable with two statistical package and two online data archives.



### 3.3.2 Interoperability

The system will need to be interoperable with a wide range of data exchanging protocols. Data can be in different data formats (XML, JSON) and be served over different communication protocols (HTTP, RPC, sockets).

To narrow the thesis, we concentrate on Fitbit and RunKeeper. As mentioned in chapter 2, data from Fitbit comes in JSON and XML format and from RunKeeper in JSON format. System shall need to handle a degree of interoperability with the web services, supporting at least one data format and communication protocol for each service.

### 3.3.3 Dependency and Availability

The system is dependent on the data archives it supports. For web services, they are out of our control, thus creating a high degree of dependency upon the services. The API, including data exchanging format and communication protocol, may be changed at any time and even taken down. Both Fitbit and RunKeeper are currently free services to integrate against. There is no Service Level Agreements (SLA) meaning that no minimal uptime, maximum response time and availability can be expected.

As mentioned in section 2.3.2, Fitbit limits the number of requests to 150 per hour per access token. This sets a limit of for availability for requesting new data when the limit has been reached. Storing data will provide a fault tolerance when the limit has been reached. RunKeeper has not stated any rate limitation. Both RunKeeper and Fitbit are big cooperations that are likely to invest money in having fault tolerance and providing high availability.

The back-end of the system may be taken down by receiving too many computational requests. The back-end will then become unavailable for clients. A typical technique to handle this is through replication having the back-end deployed at several places. Other faults not being able to reach the back-end is network partitioning. In the thesis it is left out for future work to achieve high availability and fault tolerance.

### 3.3.4 Security and Privacy

The system shall preserve athletes privacy. Accesses to the sensitive data have to be secure. Potential security breaks will identify athletes and their medical reports. Privacy and security is therefore a must for the system. Executing code received from third parties exposes a lot of security risks. In [40], they introduce four aspects to take into consideration for Remote Evaluation (REV) servers:

- *Authentication*: Client needs to authenticate him selves
- *Secrecy*: Information on the server is secret
- *Availability*: Executions are not hindered by malicious requests
- *Integrity*: The information located on the server should be preserved.

Clients shall be authenticated upon requests and the integrity of analytic functions shall be verified, to ensure they are not tampered with. The thesis will not focus on the security threats secrecy, availability and integrity (SAI) when running untrusted code. A way of dealing with this is through sandboxing, isolating the executions, limiting computational resources, and access to the file system. These three aspects are considered future work.

As mentioned in the previously chapter OAuth is typically utilized as the authentication protocol for web services. Fitbit and RunKeeper uses OAuth therefore authentication upon interactions with their Application Programming Interface (API) is required. Every request must contain the access token. Exposing the access token gives away access to the athletes data. Therefore the communication channel between the runtime and the web services has to be encrypted with HTTPS to prevent eavesdropping. Fitbit uses OAuth 1.0/a where requests are signed by the access token secret. It is therefore not as exposed as OAuth 2.0, as it has two secrets.

Although security and privacy is important, there may be parts of the system where security and privacy issues are discovered that will be considered as future work to limit the scope of the thesis.

### 3.3.5 Performance

The performance is directly linked up what the analytic function do. Therefore performance is not a criteria set. The system will be developed as a

proof of concept, giving a sense of latencies that is to be expected for analytics on online data archives, and if is feasible in terms of performance trade-offs. No specific performance constraint is therefore set. Latency for fetching data from the online data archives relies on several aspects. The server and the online data archives may be in geographical different locations, like Europe, west or east coast, thus increasing the round trip delay for communication. Then there is the latency for processing the request that may diverse out from load at the web services.

The system shall try to hide the limitations for interacting with the online data archives. The prototype will be a base for further investigation of how to improve efficiency and understand design trade-offs.

### **3.3.6 Scalability**

The thesis will not focus on providing scalability. Although the system will potentially be used by large amount of AP concurrently that will generate traffic load and computation requests. How computational heavy requests are unknown for our system in concern of resources required. A sandbox limiting processing power available for an analytic function can be a way of controlling processing power.

### **3.3.7 Accessibility**

The system shall be accessible from a large set of clients. Running analytic functions shall be by design possible from various clients with potential limited resources available. Clients that shall be supported ranges from normal Personal Computers (PC) to light clients like mobile devices with limited resources. A study shows that people are using Internet from mobile devices more than PCs [33]. Requesting analytics and receiving output shall not be by design platform dependent to for example Windows desktop machines only.

### **3.3.8 Usability**

Requesting data from the online data archives shall be easy from the statistical tools. It shall not be more complex than using standard functions in the language. The language binding to the runtime shall encapsulate the

burdens of communication including serializing input/output and handling errors.

### **3.4 Summary**

In this chapter we have presented the functional requirements of the system and based on that presented an abstract architecture. In the last section we looked at constraints of the system, several metrics and criteria to base the judgment of the system.

# Chapter 4

## Design

In this chapter we present the overall design concepts for the system and discussion around them. More details of the implementation are given in the next chapter.

### 4.1 System architecture

#### 4.1.1 System models

The main architectural decision relates to how to reach the online data services from statistical analytic tools. This decision influenced the whole system design. Three main choices are considered; local code execution, a proxy design, or a mobile code approach where code is transferred to another system and executed there.

The first way of modeling the architecture, shown in figure 4.1, is based on locally executions of functions at the client where the client communicates with the online data services directly. This would be with the assumption that the client runs on a trusted computer. In this architecture, the client would receive all the data from the different online web services. This exposes a risk of losing control over fulfilling the privacy requirements for the athletes. The client computer can be exposed to intruders either by malware or physical. In addition, receiving potential big sets of data for light clients with limited storage capabilities is not optimal. Light clients will potential also suffer of low performance and reduced energy capabilities.

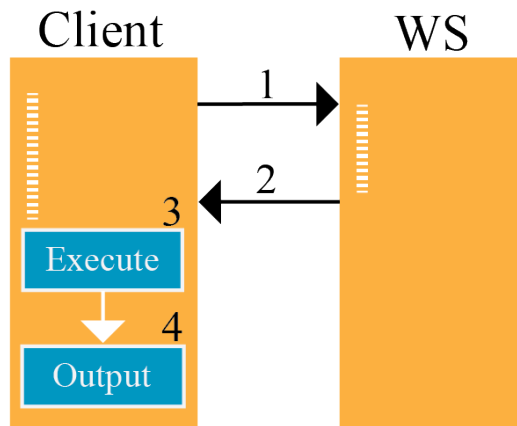


Figure 4.1: a) Client-WebService architecture.

The second architectural approach, shown in figure 4.2, is the inter-mediator approach where everything goes through an external part, a proxy. The client communicates with the proxy to get data. Extensibility through metacode can be added in this design approach by running the metacode before the data is sent to the client. This provides athletes with customizability and control over what data that is used in researches. The design complicates the process of sharing output between data analytic functions as functions are executed on the client. Having the execution on the client also is not optimal for light clients and transferring large amounts of data may be an issue also.

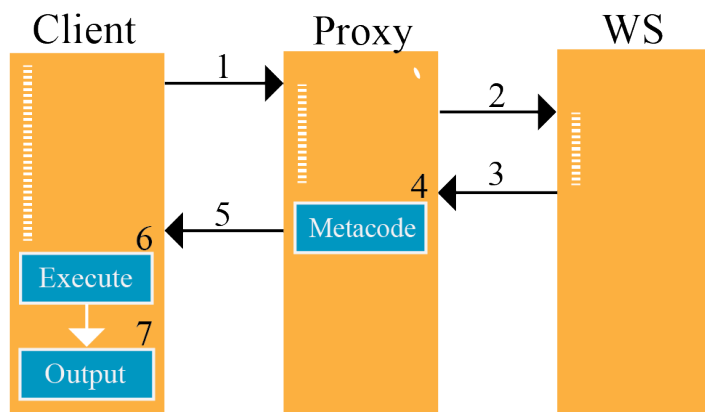


Figure 4.2: b) Client-Proxy-WebService architecture.

Based on this, the design, shown in figure 4.3, is based on a code mobility

approach. The design follows the principles of Remote Evaluation (REV), where code is transferred to a remote destination for execution. Results of the execution are returned back to the original sender. In this approach, the whole data analytic function can be executed by the runtime and therefore the runtime has full control of output produced. The analytic functions will be provided access to data archives through a runtime support via a language binding. The runtime orchestrates it all.

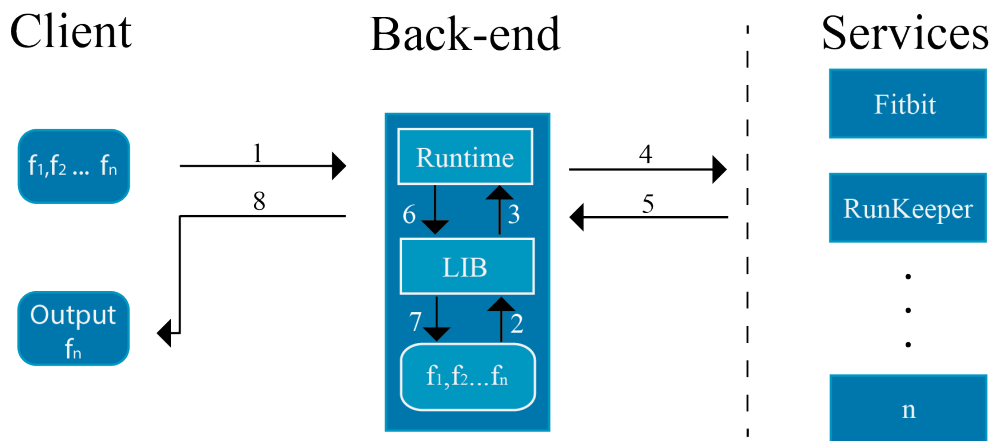


Figure 4.3: System architectural overview

Sending the code to the data instead of vice versa can improve the performance in certain scenarios [40]. Data captured by ZXY Sport Tracking System (ZXY) from a match is about 500-700 MB of storage for example. Transferring this amount of data requires high bandwidth to clients. That is not always the case, with clients using Wi-Fi networks, edge and 3G. Another thing is the computational power of the clients versus the back-end servers. Light clients may have problem with doing the computation efficient enough.

Having code mobility opens up for introducing push based result messages. In WAIF [7], wrappers around web services creating a publish subscribe system for normal web services was introduced. Push prevents clients from pulling after results wasting network resources and generating unnecessary load for the back-end when there is no updates. Code stored at the back-end uploaded by clients, can be executed every day or week with cron Unix jobs. The result can be pushed to the client(s) without any actions required by the client. Another advantage is when data is updated, either from ZXY, Fitibt, RunKeeper and other services, the back-end can be configured to run the

code pushed up by the client, and send the result back automatically. For example a graph plotting intensity from a training exercise generated after the data has been processed and ready.

### 4.1.2 Overview

Two design principles have been taken into consideration in the system design process; simplicity and generality. Simplicity for in most cases *less is more*, keeping the system maintenance and debugging simple, and potential increase the efficiency of the system instead of over-engineering. Generality for supporting extensibility to multiple statistical languages, multiple data archives and user-supplied code.

The systems main components, shown in figure 4.3, are a client, a library and a runtime. The client sends the data analytic functions to the runtime. While executing, the data analytic functions goes through the library to interact with the runtime. The library is the language binding to the runtime. The runtime can be seen as a support for the execution by handling logic for accessing the online web services. It also ensures eligibility, data violation handling through metacode and caching for reducing end-to-end latency.

The design is influenced by the broker design. In the broker design pattern, the broker has responsibility for forwarding requests, exceptions and results, in a distributed setting, where components are decoupled [39]. Adopting this design, the runtime is orchestrating data accesses, by forwarding requests between components. Sensitive data as blood pressure and hearth rate are not accessible directly and can be anonymized before being granted to the analytic functions. There can be, for instance, user IDs identifying the athlete, embedded in the response data from the online web services. Support for additional statistical analytic tools can be made by creating a library for the statistical package added, without having to modify the runtime source code. The architecture fits well with the requirement specification.



## 4.2 Back-end

### 4.2.1 Overview

The back-end accepts incoming requests for executing functions. Upon receiving a request, it validates its authenticity by computing the Hash-based Message Authentication Code (HMAC) for the request and compares it to the one in the request. The system supports chaining up multiple functions where each function  $f_n$  builds upon the output of the previous function  $f_{n-1}$ , shown in figure 4.4. The only data returned is the output of the last function. This builds on similar principles as MapReduce where the computation is staged. First, a filtering function can be run on a data set, and then a aggregation function on the result set. There is also the Sawzall programming language [34] that uses similar techniques, staging up the computation.



Figure 4.4: Functions chained where the output are sent as input to the next function

When there is multiple data analytic functions embedded, the HMAC is computed for each function to check its authenticity. Data analytic functions can be shared between APs and athletes. Athletes can give access to their data to APs via analytic functions subtracting a data set. Therefore, the client has to provide the HMAC for each function and the associated ID in our system. Any un-match of the embedded HMAC and the one calculated the execution is stopped and error is returned to the client.

While executing functions the runtime will receive requests for data from the library. The runtimes exposes a Application Programming Interface (API) for the library to interact with. A request for data is handled by either the cache or by fetching it from the web service specified in the request. Responses from the web services are anonymized before stored in the cache, and returned to the executing function. There is no functionality for requesting a specific athlete in the runtime API. Their identity is preserved and hidden from APs. However, the AP can request either one candidate or

all candidates. Candidates are athletes that are in the scope for a research project. Projects are described in 4.4.

The back-end is notified when the function has ended and has access to all output the functions has generated. This and possible errors stumbled upon during execution will be returned in the original request response.

The system supports running Python and R-scripts in chains or alone. It can easily be extended to run other statistical analytic functions as well by creating a new language binding to interact with the runtime API.

### 4.2.2 Metacode

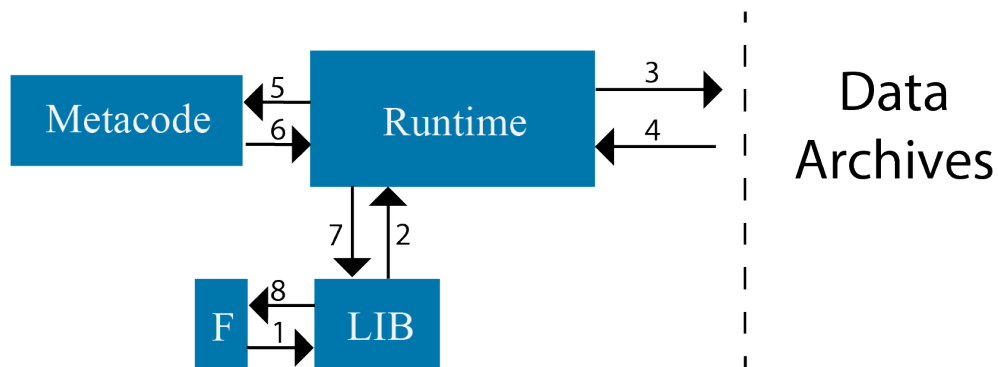


Figure 4.5: The figure illustrates how metacode is inserted into the loop when an analytic function does a method invocation. If there is any errors while running the metacode this is forwarded to the executing data analytic function following the flow 6-7-8.

As previously mentioned, the runtime supports extensibility through metacode execution for extra privacy control for athletes. Metacode extends the runtime in a dynamic manner providing dynamic system extensibility. Metacode allows you to attach code to data segments that can be run after an access to the data is made. Users can upload code and associate it with different resources like sleep, steps, activities, that is accessible in the health clouds. When an access to a resource is attempted any metacode associated with that resource would be executed. By design there can be many metacodes associated with one resource. When there is more than one metacode for a resource, they are chained and executed in the same way as with data analytic functions, where the result of the first one is sent to the next.

The metacode can throw exceptions, for example on a privacy violation for a data request wanting data generated a year ago, that will be forwarded to the running analytic functions by the runtime. The analytic function can then decide on an action based on the exception thrown. The original data requested will not be returned when exceptions are thrown. Only the exception.

### 4.2.3 Cache

A cache managed by the runtime is introduced to prevent triggering the rate limit for web services like Fitbit and providing availability of data when services are possibly down. A cache also helps to try to hide the end-to-end latency for fetching from online web services. The cache has a simple key-value design. Providing a key gets you the value stored with that key. This provides fast look-ups, preventing, for instance scanning lists to find cached values. A response from the web service is stored with 'Uniform Resource Locator (URL)/username' as key, an example is given below.

```
dev.fitbit.com/1/user/-/activities/date/2010-02-21.json/testuser
```

The runtime caches every request to the web services in the memory until the configurable max size is reached. A problem with a caching everything policy is that it may lead to inconsistency. For example requesting data from the current day. Possibly this data will change during the day as more activity is registered. There is no logic to handle this kind of inconsistency. A Least Recently Used (LRU) replacement policy is used to make room for the new objects when the cache reaches the max limit.

Another cache design that could increase the hit-rate is creating a more customized key by combining the service name, the resource requested and the date. For instance, a range query requesting data from a date, to a date, would pack all data in one response. When iterating the response each date and value is pulled out and stored as its own key-value pair. Other requests asking for a single day within that range, the data can be served from the cache, thus increasing the cache hit-rate. For further investigation into key design the access pattern needs to be analyzed.

#### 4.2.4 Anonymizer

A novel anonymizer algorithm is run on the data returned from the web services. The algorithm uses a blacklist<sup>1</sup> approach filtering out data that also appears in the blacklist. The black list contains sensitive names that can identify the athletes. It is assumed that the data is in JavaScript Object Notation (JSON) format. All keys in the data are checked by a recursive approach, drilling down in the data, by looking at new key value pairs. When a name that appears in the black list occurs the value is masked out. The returned value is the masked data.

For even more fine-grained privacy control the black list can be open for athletes to fill in themselves by for example at runtime loading the black list dynamically in.

### 4.3 Language library

The language library encapsulates functionality for interacting with the runtime, creating a language binding from the statistical analytic tool to the runtime. From the statistical tools, using the library is no different than other libraries or functions. The communication flow of the library and the runtime is shown in figure 4.6. When the Remote Procedure Call (RPC) method is called, a RPC to the runtime from the library is invoked. This call will block the processing until a response is returned to the analytic function. More details on RPC are given in 5.1.2.

Input to analytic functions that is chained together can be done in three ways:

- *RData*: For R functions an R cache object is available. The object will store whatever R object that is inserted to the cache and save it as a .RData file.
- *Stdout/Stdin stream*: Writing to Standard Output (STDOUT). The output will be piped to the next function.
- *Filesystem*: Through the filesystem storing the output in a '.txt' file. The file can be read by the next function

Support for not only R specific inputs/outputs makes us able to combine chain of scripts from other languages. The easiest and less error prone is

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Blacklist\\_\(computing\)](http://en.wikipedia.org/wiki/Blacklist_(computing))

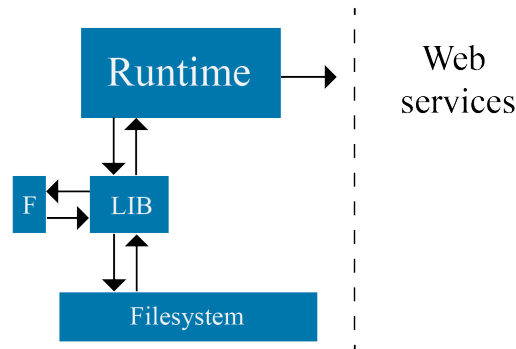


Figure 4.6: The language library interacts with the runtime to request data. In addition, the library can use the file system to store output from the analytic function.

storing output in a cache object that is serialized and written to the file system. The next function can deserialize this object and load it into the environment. Python Pickle is a similar concept that could have been used for Python scripts.

## 4.4 Access Control List and Projects

Athletes can register an Access Control List (ACL) for resources for fine-grained control of data accessibility. We introduce the concept of ACL for resources. For example, for the resource sleep, athletes can either set read only, write only or both. Only resources that are granted access to are possible to access for APs following the *principle of least privileges*. This principle is rooted in giving processes only the minimal set of privileges to be able to fulfill the task, to minify the damages it potential can do. Writing to resources is currently not supported and is intended for future use-cases. Athletes can change the ACL via the website as shown in the use-case chapter 6.

Instead of the traditional reference monitor, controlling each access to objects (resources), the ACL for each individual athlete is used to match athletes with research projects instead. For example, a research project would register that they need sleep and steps data resources. Athletes granting access to read both resources will become participants. Anytime an ACL for an athlete changes, a crawler is started to refresh participants in projects. All projects

and all ACL are scanned to map up athletes with research projects.

Upon a request from the analytic function to the runtime API, a check for eligibility is done, before continuing. The database is looked up to verify that the project have access to that resource the request tries to access. By design, all participants in the project will also have granted access to use the resource configured through ACL.

In our design model it was mentioned that willingness of participating in research projects is linked up with how it benefit each individual athlete. With the ACL design, athletes are automatically mapped up to research projects when agreeing to give away data. It exposes a risk of athletes losing control over projects and benefits. Therefore, an additional complementary confirmation mail, containing a description of the project and benefits by participating, could have been implemented as well, to prevent losing control. This would allow athletes to allow/deny projects. This is not implemented in the prototype.

## 4.5 Client

The client uses the GUI application to send files containing data analytic functions to the runtime. The client specifies which files he want to send and the order of the execution like figure 4.7 illustrates. Each file is signed to

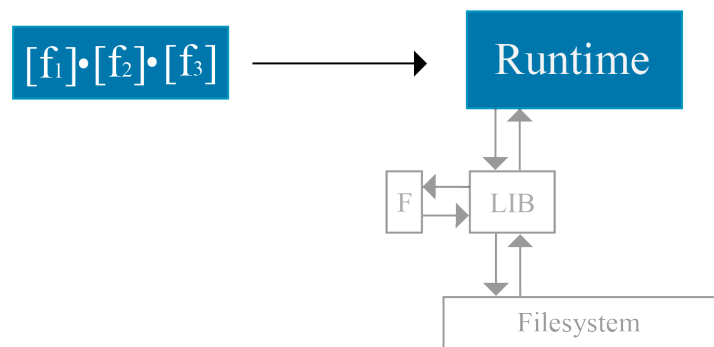


Figure 4.7: Data analytic functions are signed before sent to the runtime for execution

ensure it is not tampered with. Output from the execution will be located in the same folder as the execution file is. The client does not do any of the computations and could therefore be run at light clients without any

limitations. The main responsibility of the client is to ship the analytic functions. For this it has to be compatible with the communication protocol of the back-end, which uses Hypertext Transfer Protocol (HTTP).

## 4.6 Authentication

Interacting with web services requires authentication before getting access to the API and athletes data as mentioned in the non-functional requirements. First of all, an application is registered at Fitbit and RunKeepers website. The first time an athlete is using the system he will need to register a user giving him an ID in our system. A simple proof of concept web page handles athlete registration, authentication, uploading metacode, and configuring ACL. After registration, authentication with the web services is the next step. Athletes needs to have accounts at the web services and then grant our application access to their data. For each service, an access token is generated for the user. This makes us able to access the data on behalf of the user. The communication between the back-end and the web services is encrypted while obtaining the access token for the athlete. Fitbit uses OAuth 1.0a and will also provide an access token secret used for signing requests to the web service.

## 4.7 Storage

A database is necessary for having control over services, projects, users and credentials. The different schema's are listed in table 4.1. The service schema is bootstrapped with information about the two services at the first startup of the runtime. Rest of the schemas is filled when athletes register and projects are registered. A demonstration of the website is given in 6.2.

<b>Credentials</b>	<b>Service</b>	<b>Metacode</b>	<b>User</b>	<b>Project</b>
username	name	resource	username	name
serviceName	requestUrl	username	password	description
access_token	accessUrl	fileNames	hmacKey	resources
access_token_secret	consumerKey			participants
	consumerSecret			
	signatureMethod			
	authorizeUrl			
	callbackUrl			

Table 4.1: Database Schema



# Chapter 5

## Implementation

In this section implementation details will be given. We focus on the R implementation of the language binding. The same principles and most of the details also apply for the Python implementation.

### 5.1 R-library

The R library consists of a S4 R object that comes with methods to communicate with the runtime. It also consists of an R Cache object used to store data.

#### 5.1.1 Initiating

When the object is initiated it reads Standard Input (STDIN) to check for any input to the function from the runtime. For R functions input can be stored in an RData file on disc that will be located in the input folder. Upon initiating this file is loaded (if it exists) into the global environment for the R execution, and then unlinked from the file system. This is to prevent any other scripts in the chain to get access to the input. However, the execution can create other files in the execution folder that will not be deleted and is accessible to other functions in the chain. Another scheme for handling security and privacy issues for input and output for functions in chain is needed to prevent this as is left out of scope from this thesis.

## 5.1.2 Remote Procedure Call (RPC)

Remote Procedure Call (RPC) is the communication protocol used for analytic tools to communicate with the runtime. Communication is done over sockets using JSON-RPC that specifies methods to be invoked. JSON-RPC was chosen for its lightweight, simple remote procedure call protocol. JSON-RPC is not platform/language depended and only requires support for JSON to be supported. The runtime API can easily be extended with new methods without distributing interfaces to the client.

Upon calling the RPC library function from R, a new socket is created connecting to local host. The RPC method takes in a list containing the parameters of the method called, the method name to call and a ID for the request. It follows the JSON-RPC specification mentioned in chapter 2. The JSON object is then serialized and sent to the runtime over the socket. The response from the runtime is received on the same socket. A complication when receiving data is that the response data contains the number of bytes read before the actual JSON data. This is removed by slicing the response string before un-serilaizing the JSON response to a R-object.

On the other side, the runtime is listening for incoming connections on port 1337. Upon receiving data, the runtime unserilaizes the JSON and inspects the method field to check the request method. Currently, the runtime API has 2 methods: 'singleGetRequest' and 'multipleGetRequest' for a given URL to a web service. For both API calls it is required that the project-id is sent with. This is for the runtime to know which credentials (access tokens) to embed as authentication when requesting data from the web services. Also it is used to check eligibility of the RPC. Interactions with the web services are done in a secure channel with HTTPS.

The procedure for 'singleGetRequest' method is now described. First, the 'Project' database model is queried for the specific project by project-id. Only credentials for the requested service (Fitbit or RunKeeper) are included. A check for eligibility of the request is done. Upon a match with the requested resource and the resources the project have access to, and the request can continue. First then the web request to the web service can be invoked, but first of all the cache is checked. The cache is looked up with the key scheme described previously using the URL and the username. On cache miss, the request goes to the web service. On a response the anonymizer function is run on the result set before stored in the cache.

Further, before a response can be returned, a look up in the database is done

to find any metacode associated with the athlete and the resource the AP tries to access. The resource is found by scanning the URL to look if it contains resource name like sleep, steps or bp (blood pressure). This is a novel design scheme; if the URL contains 'sleep', the resource returned is 'sleep'. If there is any metacode attached, it is then run or the method can return the data directly. The return data is sent back to the data processing function via the socket with the format specified in the JSON-RPC specification described in section 2.1.3.

If there are errors encountered for the runtime or exceptions after running the metacode, this will be wrapped in the error object in the return JSON object. The R library will on a response check if there is any errors, and halt the execution using the built in 'stop' function. In other languages that support more specific exceptions like Python, real exceptions can be thrown. The error object in the JSON-RPC response will be 'FALSE' when there are no errors and contain an actual object when there are errors. Using 'stop' ends the execution in R, but it can be caught by wrapping the RPC method call in a try-catch clause. This opens up for specific actions out from what error that has encountered by examining the error message.

A goal for RPC is transparency for the caller that a remote invocation is happening upon calling the object method. In our library API in R this is not fully achieved since the caller uses one method for every API call.

### 5.1.3 R Cache

R supports object serializing by storing the object in a binary representation on disc. The implementation of R cache is an S3 class. The class contains a list for storing properties to be saved. Saving and loading the object is done with the built in R functions save and load. Saves stores the object as a binary file. Using load, the binary file is converted into R Object and loaded into the global environment of the R execution. A static hard coded file name is used when saving the file. This way the next R function in the chain knows what file to load. The final output of the execution of an analytic function has to be sent through Standard Output (STDOUT) and not by using the R cache.

Resource	HTTP Method	Response
/proxy	POST	Files
/static/	GET	Files
/fitbit_cb	GET	HTML
/runkeeper_cb	GET	HTML
/authenticate/:serviceName	GET	redirect
/user/new	POST	HTML
/metacode	GET	HTML
/metacode/new	POST	HTML
/acl	GET	HTML
/acl/new	POST	HTML

Table 5.1: API

## 5.2 Runtime and WebServer

The runtime is written in Node.js [45], Node from now on, is a packaged compilation of Googles JavaScript engine<sup>1</sup> engine. Its a relatively new (being released in 2009) and unproven platform, but has gained a lot of attention in the computer science community for its lightweight and efficient model. This comes from the even-driven and non-blocking I/O model. Per default, an instance of node runs in a single thread and does not utilize other CPU cores. When writing a web server with Node, you need to avoid long, blocking computations as this will block the node process for handling new incoming requests. Our runtime uses non-blocking I/O calls when accessing the database or accessing the file system to avoid this issue. Initiating computations in child processes makes it possible to do long CPU intensive computations. The root process of a child process controls standard streams and can be piped back after the execution is done or at runtime.

The runtime supports executions of Python and R code, integrates with Fitbit and RunKeeper, and it embeds a web server. It exposes a interface over Hypertext Transfer Protocol (HTTP), shown in figure 5.1. Requests are handled based on the resource Uniform Resource Locator (URL) accessed. The API is listed in table 5.1. In addition, the runtime listens for incoming connections on a socket to handle communication with language bindings.

The runtime takes use of several node packages<sup>2</sup>, including the Express

---

<sup>1</sup><https://code.google.com/p/v8/>

<sup>2</sup><https://npmjs.org/>

framework, which is a web framework.

### 5.2.1 Running data processing functions

In the non-functional requirement we stated that securing the secrecy and integrity of the server data was left for future work. It is assumed in the implementation that executions of code can be isolated and that the execution cant tamper with files on the server.

The main URL is the '/proxy' URL. The runtime expects a HTTP POST with Content Type header set to multipart/form-data. This is a content type used for sending multiple files over HTTP. In addition to the files, a Hash-based Message Authentication Code (HMAC) and a username for each file are expected to be embedded in the HTTP headers. When a request is received by the runtime, one at a time the file is read and written to disc, then executed, before repeating for the next file. For a file to be run it has to be valid. Each file must have a HMAC associated with it. The signing has to be done with the hash algorithm SHA1 and generate a 40 bytes hex string. The HMAC is computed by taking the raw bytes of the file and a secret key generated when a user is created. The secret key is generated by first generating a random seed of 20 bytes using the built in crypto module of Node. This seed is then again hashed with the SHA1 algorithm, and generates a 40-byte hex string that becomes the secret key. Invalid requests where the HMAC doesn't match are aborted. Having files signed, helps to verify that both the authentication and the data integrity of the files are valid.

Before any files are executed the runtime will create a folder for the request. This is a random named folder, equal to how the secret key is generated, that is removed after all files have executed. All input and output for the data processing functions will be in this folder. This means we can run multiple requests from the same AP or multiple APs at the same time. Input/output from executions isnt mixed up. The folder is deleted when the request response is returned.

For each incoming function the runtime will spawn a child process for execution of the file. The child processes always has three streams associated with them: stdin, stdout, stderr. Upon errors while executing a file, error messages will be received by stderr. All executions will be stopped and the error returned back to the client in the HTTP header. Input to functions is streamed through the stdin stream. This is done right after the process is

spawned. Input to functions is optional.

Output from the last functions stdout stream is placed in the HTTP header as 'stdout'. A problem is to know what to send back to the client as the analytic function may produce arbitrary output, graphical and text files. The current solution only supports graphical output in PDF file. When the processing are finished the runtime checks if there has been generated a 'Rplots.pdf' in the execution folder of the request, and inserts the file as an attachment<sup>3</sup> in the HTTP response if there is. Graphical output works both for R and Python when the output is named 'Rplots.pdf'.

An error that may occur is the stdout max buffer exceeded error. By default Node spawns the new child process with 200\*1024 bytes buffer. If this is exceeded an error is thrown and the execution stopped. The buffer proved to be too little while running experiments and was boosted to 10 kB, which was sufficient for our experiments. Another approach would be to use the 'child\_process.spawn' function instead. It returns an object with stdout and stderr streams. Then we could continuously read the standard out stream for data, and continuously start streaming the data back to the client via the network for example. Another possibility would be to stream it directly to the next function in the chain.

## 5.2.2 Metacode

An execution of a metacode is started up by starting a child process from the runtime. The data to evaluate is streamed in as stdin to the process. The output is written to stdout. A small library in Python standardizes the process of reading input and output for all metacodes. Only metacode written in Python is currently supported. The main issue with implementing metacodes is how to catch exceptions that is thrown by the metacode. An external Python script runs the metacode in a try-catch clause like shown in the code snippet below.

```
try:
    execfile(sys.argv[1])
except, e:
    sys.stdout.write({errorInMetacode : True, message: e.message})
```

The metacode filename is given to the script as a system argument when the script starts up. By using the built in Python command 'execfile' the

---

<sup>3</sup><http://expressjs.com/4x/api.htmlres.download>

file can be executed as the code was embedded in the script. This way we can catch the exception thrown by the metacode and serialize the exception into a special JSON object containing the exception description. Language wrappers can be built to transform the exceptions description to built in, real exceptions in the statistical languages.

When the metacode is finished running the runtime is notified. The runtime will serialize the output from the metacodes stdout to JSON and check if it contains an error key 'errorInMetacode'. The error key signals that an exception has been thrown. There may be scenarios where the output cant be serialized and an exception will be thrown in the runtime. This, however, is caught, and it is assumed that everything is ok and no exceptions have been thrown by the metacode, as this would have been caught with the external Python script.

An example of a metacode is given below. It iterates through the result set containing steps per day. If the steps are more than 10 days old an exception will be thrown.

```
steps = data['activities-steps']
now = datetime.datetime.now()
for step in steps:
    date = datetime.datetime.strptime(step['dateTime'], '%Y-%m-%d')
    if((now - date).days > 10):
        raise Exception("Data expired")
```

Uploading of code is done through the website. Code will be stored in the file system in a special folder for the athlete under his username, like 'public/uploaded/username/hexstring.py'. The filename is a random hex string generated the same way as the secret key mentioned earlier. The root folder for the athlete is created when he register in our system. The metacode data is then inserted into the database and will be looked up later and run when data for the athlete is used. The folder containing the metacode is assumed to be only accessibly by the runtime, secret for executing functions and integrity preserved for privacy reasons as stated in the section 3.3.4.

### 5.2.3 Cache

The runtime cache is implemented as a JavaScript object using the prototype property to add functionalities like attributes and methods to the object. It exposes *get*, *put*, *del* and *clear* methods. The cache data container is implemented with a standard JavaScript dictionary to match the key value

design chosen. To handle the least recently used policy a list is used to keep track of the least recently used items. The first item in the list is the least recently used and the last, the most recently. An additional dictionary is used to know where each key is in the list to avoid scanning through the list every time a key is used. A single lookup in the dictionary and we know the location of the key in the list, this item can be deleted from the list and placed last in the list. The extra dictionary adds some extra meta-data, but is small compared to the cache data it serves, as there can be large datasets for each key. The size limit of the cache is by the number of keys.

### 5.2.4 Anonymizer

A blacklist of all field names that is considered to be identifying the user is first of all created; `userId`, `memberSince`, `fullName`, `displayName`, `dateOfBirth`, `height`, `gender`, `country`, `weight`. All keys checked are transformed to uppercase to match the keys in the blacklist. The anonymizer algorithm function receives a JavaScript Object Notation (JSON) object with a set of key value pairs. Keys that are in the blacklist are masked by writing over their corresponding value. This prevents the algorithm of scanning through every key. The end result is the masked, anonymized JSON object.

## 5.3 Client

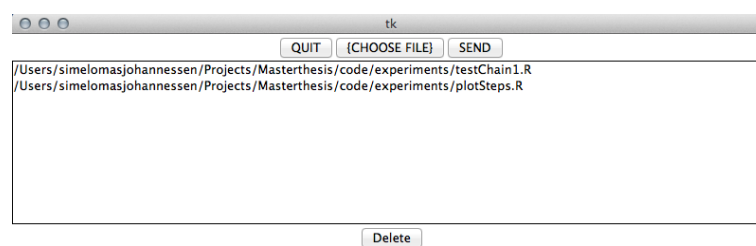


Figure 5.1: GUI application. The file listed first is the first to be executed

The client is implemented in the Python library Tkinter<sup>4</sup>. This is small GUI application used to send files to the runtime. When the 'send' button is pressed, an HTTP POST is sent to runtime containing the files the user has

<sup>4</sup><https://docs.python.org/2/library/tkinter.html>



specified. In addition each file is signed using the clients secret key with SHA1 hash algorithm.

In the HTTP response from the runtime the output from the last function executed is wrapped in the HTTP header field 'stdout'. This output is written to a '.txt' file in the folder the application was started in. If the last function produced graphical output it will be stored in 'outputFile' field in the HTTP header. This will be read and written to a PDF file, the only graphical output currently supported.

## 5.4 Database

The database is chosen is MongoDB. Chosen for its ease of use from Node. It is also well documented, that helps when running into problems. MongoDB interaction is done through the Mongoose module. It supports direct insertion of JavaScript Objects that are mapped to database schemas. Every schema stores only strings. This is the only data type used except from files. Some fields have indexes to secure uniqueness. This includes 'serviceName' in the Service schema, 'username' in the User schema. A thing to notice is the Project schema that embeds a reference to the credentials schema in the participants field. A normal query on the Project schema only returns the MongoDB ID of the credentials documents. MongoDB ID is a unique ID generated for every document. To embed the credentials documents, and not only the ID, a populate query is run. MongoDB will then do a operation like the join operation in Relational Database Mangement System (RDBMS) where the credentials documents are fetched and inserted to the participants field, replacing the ID. It is also possible to do filtering on the credentials documents that is embedded. For example a request wanting only Fitbit credentials for a project will only embed credentials for Fitbit by filtering on 'credentials.serviceName' = 'Fitbit'.

## 5.5 Webpage, authentication and users

A simple web site in HTML provides the interface for athletes to create users, authenticate themselves with Fitbit and Runkeeper, upload metacode and configure Access Control List (ACL). Fitbit and Runkeeper is bootstrapped by inserting all the details like request URL, access URL, consumer key, consumer secret, signature method, callback urn and authorize URL, into the

database. This details are found at the services website when an application is created. The runtime has callback URLs for Fitbit and RunKeeper services to redirect the user to after granting access for the application. For example RunKeeper will redirect the user from the RunKeeper servers to the back-end URL `'runkeeper_cb?code=22e5c5f8d63d4001b23e30bf434bad0f'`. Code is a temporary token used in the last exchange before receiving the actual access token between the back-end and the web service. Everything around OAuth is handled by the Node module `node-oauth`<sup>5</sup>.

When the users fills in the registration form and submit it, a cookie for storing the users username is created. The browser will send this on every request. Cookies enable us to save state across many requests over the stateless HTTP protocol. When the user navigates to the access control page, the server fetches the ACL for the user based on the username stored in the HTTP cookie field.

Upon authenticating with Fitbit the server has to keep track of the temporary access token secret as specified in the OAuth 1.0/a protocol. Again the users cookie is used. When the server gets a callback Fitbit this can be extracted from the cookie. The authentication flow follows the standard as described in section 2.2.

Every request to the web server needs a cookie with a username or they are redirected to the user sign up page. The exception from this scheme is the callback URL's, the proxy URL used by the client application, user sign up page and any static files like HyperText Markup Language (HTML), Cascading Style Sheets (CSS) files.

The secret key used for signing requests is generated upon user creation following the same algorithm as mentioned in 5.2.1. This has to be shared to APs and hidden for strangers.

Using HTML and the Handlebars<sup>6</sup> view engine create the actual web pages. The view engine handles caching of files to save I/O operations. To achieve dynamic content on the web pages, data loaded from the database is inserted into the HTML document before being returned to the web browser. A main skeleton HTML page is the basic for every page, containing the header navigation elements. Out from the page that is visited a different HTML document is inserted to the HTML body tag of the skeleton HTML page. An example from the ACL page how access permission are inserted dynamically are shown in figure 5.2

---

<sup>5</sup><https://github.com/ciaranj/node-oauth>

<sup>6</sup><https://www.npmjs.org/package/express3-handlebars>

```
<ul class="list">
  {{#acIs}}
  <li>
    <h3 class="resource">{{resource}}</h3>
    <p class="permission">{{permission}}</p>
  </li>
  {{/acIs}}
</ul>
```

Figure 5.2: HTML of the Access Control List page. Curly brackets define input of dynamic data. In this page a list of ACLs is inserted, where each object contains a resource and permission key-value object

The web page part of the runtime could have been a own service running somewhere else and not a integrated part of the runtime as present. However, abstracting it away from the runtime doesn't require much effort because of the module based code implementation.



# Chapter 6

## Use-Case

This chapter gives a demonstration of the system including showing the output of four analytic functions and a demonstration of the web site.

The data in the examples are real data fetched from Fitbit and RunKeeper. Users in the examples have used Fitbit Flex since 01-01-2014. The data from RunKeeper is data from runs recorded with the RunKeeper application for Android.

### 6.1 Analytic functions

#### 6.1.1 Sleep and Steps

The first function plots the link between sleep quality and daily activity. It is written in R and uses Fitbit data as source. The hypothesis is to see if with more activity during the day, the sleep quality increases.

Plots like this can be evaluated by human interpretation and visual analyses of the plot. Any clustering of data can confirm the hypothesis one way or another. Errors in the data occur, as Fitbit Flex requires manual registration of sleep. You tap the device when you go to sleep and when you wake up. For instance, a result where the restless duration is above 600 minutes is most likely a result of forgetting to end the sleep sequence.

The output is shown in figure 6.1, 6.2, 6.3 and 6.4.

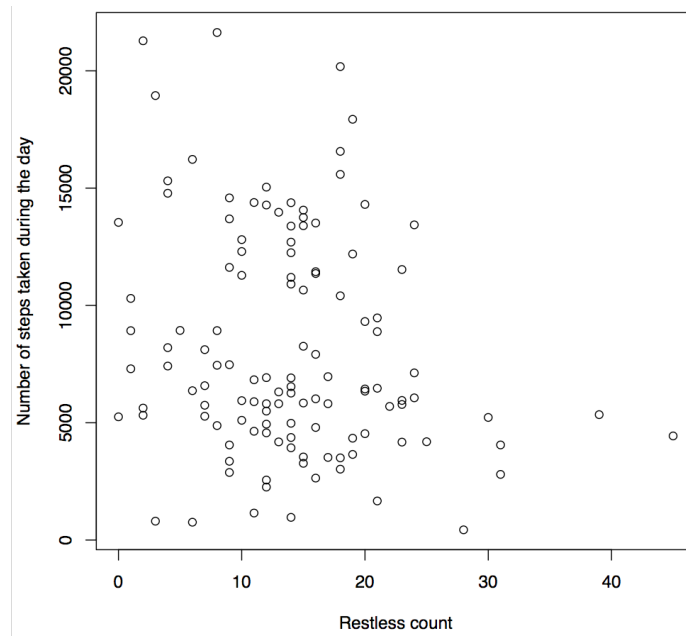


Figure 6.1: Plot of number of steps taken and the number of times restless during the following night sleep.

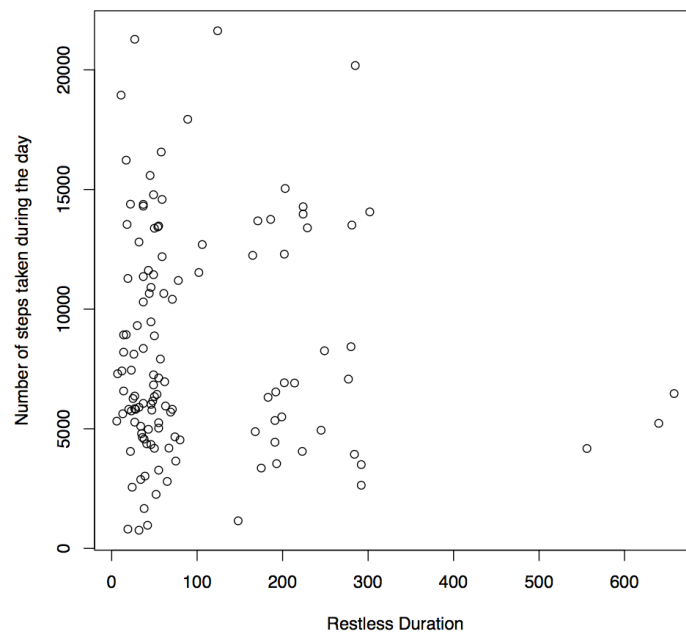


Figure 6.2: Plot of number of steps taken and the amount of time in minutes restless during the following night sleep.

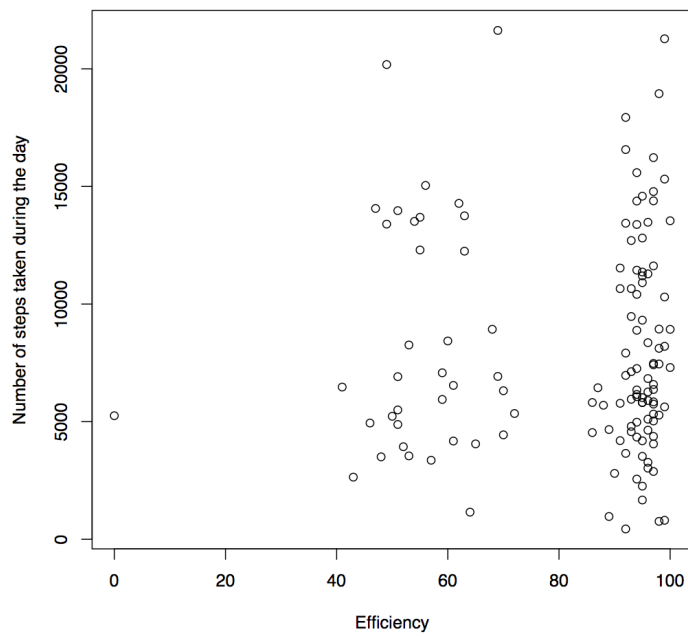


Figure 6.3: Plot of number of steps taken and the efficiency of the sleep the following night sleep. Efficiency is a percent calculated by Fitbit

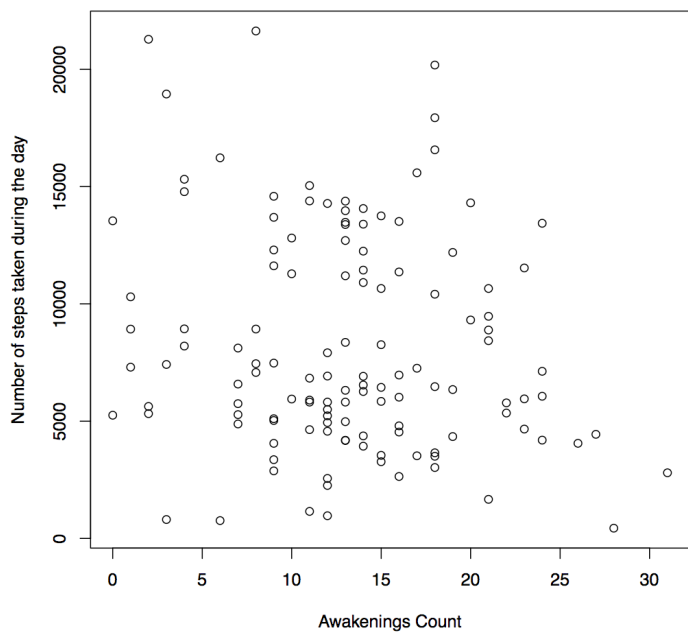


Figure 6.4: Plot of number of steps taken and the number of times awakened during the following night sleep.

### 6.1.1.1 Result

Out from the results, it cant be concluded with anything. There is no clear clustering pattern in any of the plot. If anything, the plot confirms that there is no such link between activity and sleep. In addition, quality of the data is probably not the best with users forgetting to end the sleep when awakening.

### 6.1.2 Steps

The second analytic function is a simple dynamic plot of the number of steps taken over a range of days written in R. Data is gathered with Fitbit Flex. Days where 10000 steps are reached is marked with green and days below are marked with red. The output is shown in figure 6.5

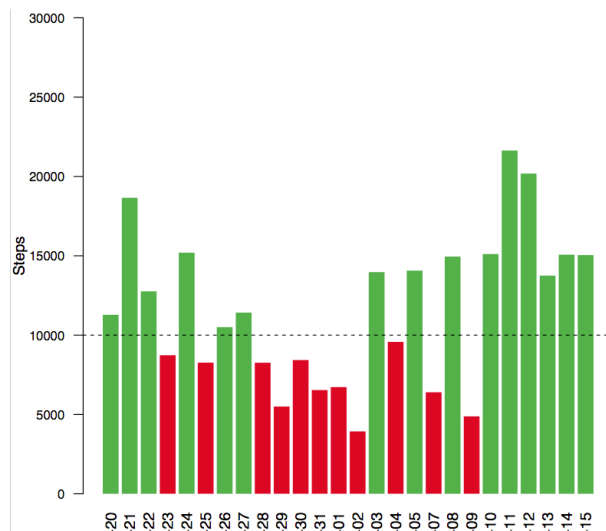


Figure 6.5: Daily step count over a range of dates. The date is hidden for privacy reasons.

### 6.1.3 Running

The third analytic functions written in R uses data from RunKeeper. The data is a run recorded with the RunKeeper application for Android, fetched from the RunKeeper healthgraph API. The output is shown in figure 6.6 on the next page.



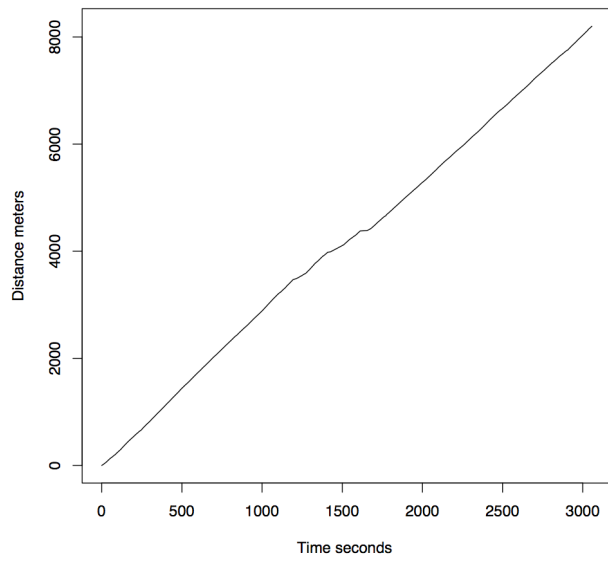


Figure 6.6: Plot of time in seconds against the number of meter done during a run.

### 6.1.4 Sleep cycle

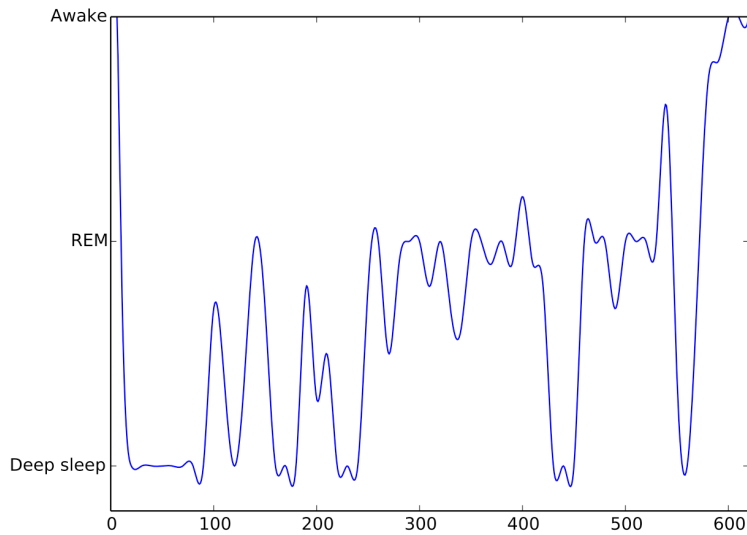


Figure 6.7: Plots the stages of sleep during the night. In the x-axe the time asleep in minutes. In the y-axe the different stages: Awake, Rapid Eye Movement (REM), deep sleep.

The fourth analytic function uses data from Fitbit to plot the stages of sleep

during night. It is written in Python. The calculation of which sleep stage you are in is based on movement over a period of time. An assumption here is that movement is linked up to sleep stages. A lot of movement indicates that you are an awake, lighter movement in Rapid Eye Movement (REM) stage and little to non-movement in deep sleep. A problem here is the granularity of the data Fitbit provides on their sleep. Each minute has a value associated that variate between 3 values: 1, 2 and 3. The values indicates awake, restless and sleep. With higher data granularity a better plot could have been made. The output is shown in figure 6.7.

## 6.2 Web site

[Home](#) [Services](#) [Access control](#) [Upload metacode](#) [Images](#)

### Create a new user

Enter username :

Enter Password :

Figure 6.8: User sign up page. User fills the sign up form to create a user

In this section we give a demonstration of the web page. First of all, the user needs to sign up before any other page can be visited. Signing up is done by filling out the form and submitting.

### 6.2.1 Services

In this page all web services the system is linked up to are listed. Clicking on one of them starts the authentication process. The user is redirected to the requested web service website to authenticate our application to use their data.

## Services

- [Fitbit](#)
- [RunKeeper](#)

## Authenticated

- Fitbit

Figure 6.9: Services page. All web services available are listed here

### 6.2.2 Access Control List

Shown in figure 6.10, is the Access Control List page for adding or removing access to resources. Only resources that are listed can be used by APs. New resources can be added by filling out the form at the bottom of the page. The user can update an existing resource by overwriting the old. Currently, there is no way of deleting a resource.

#### Access Control List

Only listed resources can be accessed

<b>sleep</b>
Read
<b>steps</b>
read
<b>bp</b>
Read

#### Create or update new access control permissions

Enter resource :

- Read only  
 Write only  
 Write and Read

Figure 6.10: Access Control List page. To create new resource the user enters the resource and sets what access permission it should have.

### 6.2.3 Metacode

Uploading of code to be associated with resources is done in the 'Metacode' page. The user chooses a file to upload and what resource to associate the metacode with. Examples of resources are sleep, steps, bp (blood pressure) and activity. On submit the file is uploaded to the back-end.



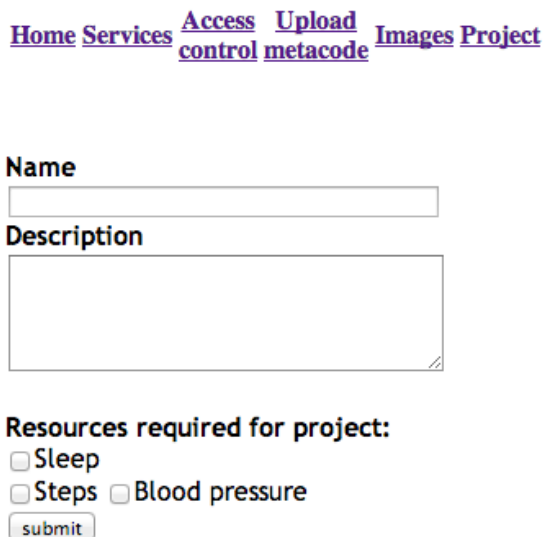
Resource to associate with (SLEEP, STEPS):

No file chosen

Figure 6.11: Metacode

### 6.2.4 Project

To create a project, the user fills in the form with name of project, description and which resources that it requires, like shown in figure 6.12. On submit the project registration is sent to the back-end.



[Home](#) [Services](#) [Access control](#) [Upload metacode](#) [Images](#) [Project](#)

**Name**

**Description**

**Resources required for project:**

Sleep  
 Steps  Blood pressure

Figure 6.12: Simple project registration. Only three resources is listed for demonstration purposes

# Chapter 7

## Evaluation and Results

This chapter presents methods used to evaluate the system; results collected evaluating the system and a discussion around the system including non-functional aspects. The discussion of the results is given right after the result is presented.

### 7.1 Methods

The goal of the experiments is to validate the system sustainability for doing statistical analytic, and the architectural and design choices made. To do this we conduct several experiments and evaluate system metrics; latency, CPU utilization and memory consumption.

The CPU utilization relies much on the analytic functions executing as it may do arbitrary CPU intensive computations, and it is without the control of the runtime. Memory consumptions depend on how large the runtime cache size is, and the analytic functions memory usage. Therefore, the most interesting metrics to measure is the latency. This is done for RPCs, end-to-end, anonymizer and metacode:

- *RPC*: Latency is measured from the RPC call in the analytic function to a response is received, measured by using `Sys.time` in R.
- *Metacode*: Latency is measured from the metacode is initiated in the runtime and to a response is received when the subprocess finish, measured by using `console.time` in Node.

- *End-to-end*: The latency from pressing send in the client application to a response is received, measured in Python using `time.time`.
- *Anonymizer*: The anonymizer algorithm latency from calling the function until a results is returned, measured using `console.time` in Node.

The experiments are done to measure what latencies that can be expected when using the system for statistical analytic tasks, and evaluate if this is feasible. Two persons Fitbit account have been active since 01-02-2014, generating data, both sleep and activity. Both these accounts is integrated in the system with credentials for Fitbit and added to a project. The test project requires access to steps and sleep resources, which both test user accounts have granted access to.

To evaluate CPU utilization the terminal command `sar` in Ubuntu was used; `sar -u 0.3 100t`. It will show the CPU utilization every 300 ms 100 times. The CPU utilization is defined as the amount of CPU a given process uses, measured in percent of total CPU capacity.

## 7.2 Experiments and Results

### 7.2.1 Experiments Setups

The system runtime is deployed on DigitalOcean<sup>1</sup> cloud, which is a simple cloud hosting service. Our test machine runs Ubuntu 14.04 32bit. The virtual machine has the following specifications:

- 2.7-3.1 GHZ CPU
- 20GB SSD Disc
- 512 MB ram

It uses only one core, but can be scaled up on demand to use more Cores. The machine is hosted in their New York 2 server park. The data round-trip delay time is 122 ms from the client in University of Tromsø to the test server. Both the runtime written in Node and the database MongoDB runs on this virtual machine. All experiments have been run 20 times to get an average performance measurement, and hide skewed results.

---

<sup>1</sup><https://www.digitalocean.com/>

## 7.2.2 Experiments

### 7.2.2.1 Experiment 1

The first experiment (1), consists of one data processing function. It starts by fetching all the sleep records from Fitbit since 2014-01-01 and 45 days forward. Sleep has to be fetched a day at once, a limitation in the Fitbit API. Total, 44 Remote Procedure Call (RPC)s is required. In addition, the data processing function also fetches the steps count for the same days. This can be done with one RPC. It then combines the data having the daily step count linked with the number of times restless the following night. Only one Fitbit users data was used in this experiment.

The metacode in the experiment scans through the data set, checking that the date is not older than two years, and throws an exception if older. The analytic function at last plots the result in a graph that generates a PDF file that is returned to the client. The runtime cache was off during this experiment meaning every data request had to hit the Fitbit servers. The results (1) are as following measuring latency:

Algorithm	Min ms	Max ms	Avg.
Anonymize	0	38	1
GET Fitbit	274	5385	368
Metacode	81	87	83
End-to-End	20189	25998	23093

From the results above, requesting 45 days of sleep and steps data from Fitbit service has a average latency of 368 ms. In average for 46 requests that is required in the experiment, the runtime will have to wait 16928 ms on response from Fitbit. The reason for this is the sequential design of the RPC in the R library. Each request to the runtime is done one at a time, sequential, taking large fractions of the overall execution time. Hiding latency for communicating with the online web services was a goal for the system. The network latency in this experiment dominates the overall execution time.

A possible approach to speed up RPCs is by introducing threads. It will prevent blocking the execution of the analytic function while it is waiting for a response. Threads can be a bit hassle and painful, and to evaluate potential speed up another approach was chosen by complementing the runtime API with a new method. The new method spawns requests from a date to a date simultaneously. Network calls are seen as I/O in Node and runs in

background until a result is return. In average 45 days of sleep data could be retrieved in 2805 ms using the new API method, compared to the previous approach that took an average 16560 ms for requesting the same data, a decrease of 83 percent.

In the experiment metacode is only run once, when requesting the daily number of steps. Starting a new subprocess runs the metacode. Processes are cost heavy having to allocate resources by copying memory and creating entries in process table. In addition, the metacode can do arbitrary long computations without any restrictions, and therefor it must run in an own process to prevent blocking Node's event loop. Currently, metacodes have full access to the Operating System (OS). We can assume athletes do not upload malicious code, but others can gain access to their account and upload dangerous code. A solution to handle this is by code inspection before approving the metacode.

The anonymize functions execution time varies out from the size of the data set. Days where there is no sleep data recorded it finished in 0-1 ms. The data set that required the longest time run finished in 38ms. This data set contained sleep data for 582 minutes.

Running experiment 1 with all data already located in the cache, the end-to-end latency was in average 2045 ms:

Algorithm	Min ms	Max ms	Avg.
End-to-End	1838	2443	2045

As expected, minimizing requests to Fitbit servers is key reducing the end-to-end latency. The end-to-end latency went down 91.1 percent when all data was cached compared to the sequential. This suggests that the system should store more data to be more efficient. The cache in the runtime stores data now, but memory is precious and small in size compared to magnetic discs. A suggestion is to store more data on disc and use pre-emptive crawlers to speed up executions of analytic function. A little longer overhead for accessing data can be tolerated especially taking into consideration the latency for fetching data from the online data archives.

The other data archive the system integrates, RunKeeper, shows a longer average latency for requests to retrieve fitness activity for a user, 'https://api.runkeeper.com/fitnessActivities/id'. Retrieving a response for this URL took an average of 1240 ms from the server.

Figure 7.1 shows the result of multiple clients trying to run experiment 1 in parallel. All data requested was cached. As the results shows the latency



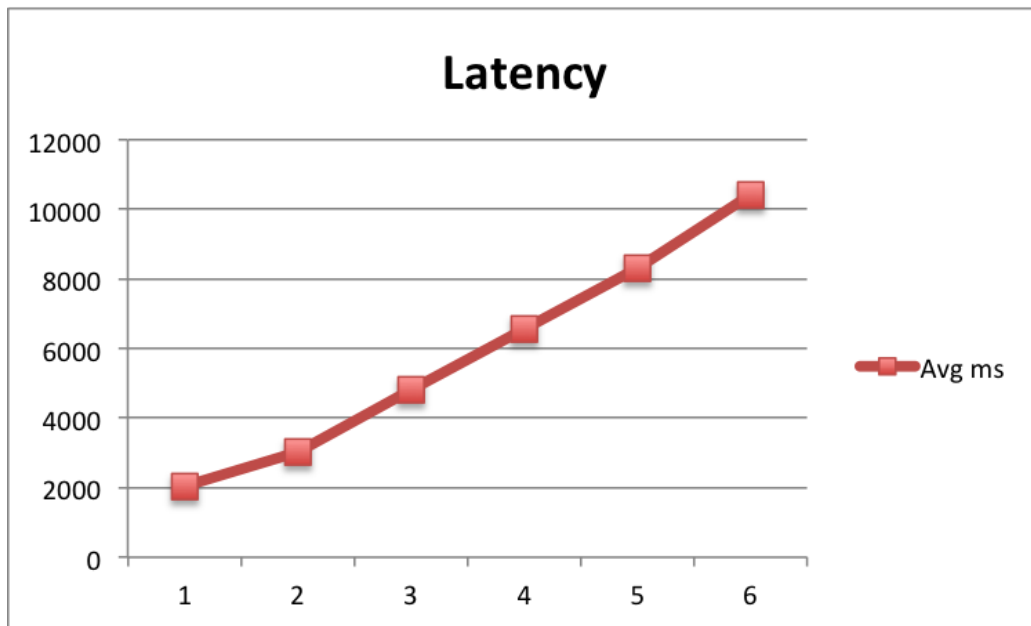


Figure 7.1: Latency in ms when multiple clients runs experiment 1 in parallel

increases approximately linearly when more clients are run simultaneously. With 1 client an average of 82.43 percent of the CPU is utilized while running. Already with 2 clients running in parallel, 100 percent of the CPU is utilized. This is most likely because of the heavy deserializing/serializing of JSON that is known to be CPU intensive. JSON is our data format for data-interchange between processes and the Fitbit data comes in JSON format, which requires to be deserialized before the anonymize function can be run.

The test machine setup has only one core. The runtime does not utilize more than one core running in a single thread. Using the cluster module<sup>2</sup>, Node can take advantage of multi-core systems.

### 7.2.2.2 Experiment 2

To evaluate the RPC procedure, a hard coded JSON of 29.1kB was returned on a RPC call to the runtime. In average it took 42.4 ms for the RPC call to finish in R. In Python this was significant faster returning in average 4.9 ms. To have something to compare with, the same JSON response was returned over the socket without deserializing/serializing overhead. On average, the latency was 3.79 ms.

<sup>2</sup><http://nodejs.org/api/cluster.html>

Procedure	Min ms	Max ms	Avg. ms
R: JSON-RPC 29.1 kB	34,7	143	42.4
R: JSON-RPC 37 bytes	3.0	4.3	3.5
R: Socket 29.1 kB	3.33	6.49	3.79
Python: JSON-RPC 29.1 kB	3.4	16.6	4.9
Python: JSON-RPC 37 bytes	1.8	5.5	2.5

Our Inter-process Communication (IPC) is based on TCP over sockets. Time consumers for normal socket communication include establishing socket connection, and transmitting the data over the network interface. In additional, when including the JSON-RPC protocol, overhead for serializing and deserilazing the request/response. A good protocol language is vital. It should be efficient to encode/decode, and be compact in serialized form. Efficiency lies on the implementation of the JSON serializer, both in Node and in the analytic tools. From the results we can see that the R JSON serialize library RJSONIO is significant slower than the native JSON serializer in Python 2.7.2. There are several approaches to make JSON faster to serialize/unserializer and more compact. Attempts of this include BSON<sup>3</sup>, BJSON<sup>4</sup> and UBJSON<sup>5</sup>.

## 7.3 General Discussion

In this section a discussion of if our goals and other functional/non-functional specifications was achieved.

### 7.3.1 Extensibility and Interoperability

One of our main goals was that the system should support dynamic extensibility at runtime. We introduced metacode to cope with this problem. Athletes can upload code and associate it with resources like sleep, steps, fitnessactivities and bp. The code is guaranteed to run every time the resources are accessed by a analytic function. Multiple metacodes can be associated with a resource and everyone will be executed. Exceptions thrown will be forwarded to the analytic function.

---

<sup>3</sup>[www.bsonspec.org](http://www.bsonspec.org)

<sup>4</sup>[www.bjson.org](http://www.bjson.org)

<sup>5</sup>[www.ubjson.org](http://www.ubjson.org)

In our interpretation we stated that the system should be extensible to more statistical packages and data archives. Providing support for RunKeeper and Fitbit for data archives, and support for both R and Python proves this. The module-based code makes it easy to extend to use other data sources like relational databases. Creating a new Application Programming Interface (API) method in the runtime, taking in the query to be run on the database, could provide this. Extending to more statistical analytic tools requires a new language binding supporting JSON-RPC over network sockets. Requirements for the statistical tool to be supported is being able to receive input from Standard Input (STDIN) or by reading a file in the file system, and be able to share output via Standard Output (STDOUT)/filesystem.

A limitation for our goal of interoperability is the anonymize function running on the returned data. The implementation of it assumes the response from the data archives is represented in JSON. Running the current implementation on XML data would not work. However, the system supports at least one of the file formats (JSON) the data archives uses that was required as stated in the non-functional requirements.

### 7.3.2 Security

Introducing Remote Evaluation (REV) has its security risks that need to be addressed. A recap of the four aspects introduced in the non-functional requirements to take into consideration for REV servers:

- *Authentication*
- *Secrecy*
- *Availability*
- *Integrity*

We stated that authentication would be the only one being focusing on achieving in the prototype. Authentication is secured with requests having to contain a Hash-based Message Authentication Code (HMAC). Invalid requests not signed by an AP with a secret key are aborted. The code in the request will not be executed.

Secrecy, availability and integrity were stated as future work. In our system design it is assumed that each data processing functions runs in a sandbox. A sandbox is used to separate running programs. Programs running in a sandbox have controlled set of resources available. Limited memory space,

little to no disc access and ability to inspect the host. Typically, sandboxes are used where you have to run untrusted code. For R, the RAppArmor [32] package can be used, which is essentially a sandbox for Unix to run R script in by enforcing security restrictions on processes.

In concern to availability an analytic function may do computation heavy processing on purpose to slow the system down. Solutions exist to limit the processing power of a process for example by lowering the priority of the process, or specific setting a upper limit of CPU utilization for the process.

In terms of integrity a security risk is that analytic functions accesses each other input and output. This can violate athletes privacy. Currently, executions have full access to the file system. It also means that metacode stored for each athlete is accessible and can in theory be modified violating the integrity and secrecy. A possibility to prevent this is to restrict the process to only have access to its execution folder. In Unix 'chroot' does this restricting the process to access files outside the directory tree.

### 7.3.3 Privacy concerns

A problem with sensitive data and research projects is to preserve privacy in long term when possible new algorithms is discovered that can point out individual participants. Our solution with dynamic operative consent through Access Control List (ACL) and metacode extensibility helps the user controlling changing environments. Potentially, new data sources can be added providing higher data granularity, and thus being able to identify athletes. With metacode extensibility, athletes have the option to ensure that privacy is protected while environments are changing.

Using big companies devices and infrastructure have its advantages; they take care of consumer support, software upgrades and web portals for managing and displaying data. However, there is privacy concerns to think about. One thing is that our system can preserve the privacy of athletes data as good as possible. Another thing are the companies behind the services used.

Concerns are raised [27] [28] that data collected about consumers can be sold to other companies using it for example advertisement targeting. In Fitbits privacy policy<sup>6</sup> they state that they can make personal information collected from users available to strategic partners. Essentially saying that they can

---

<sup>6</sup><http://www.fitbit.com/uk/privacy>

sell your personal information to other companies. Internet users in Europe have also raised concerns. Stated in the report Attitudes on Data Protection and Electronic Identity in the European Union [9], 70 percent says that they are concerned that companies may misuse personal data. Personal Data Vaults (PDV) [31] is an approach to control personal data where the user retains ownership to the data.

In the system an anonymize algorithm is run for every request to the web service before stored in the cache. A problem with it is that it is only as good the blacklist. The algorithm goes by name of the keys in the response from web services. A possibility is that a web service uses a shortening of a word or a acronym. Semantics may also diverse where a key is meaning something else than our interpretation. A suggestion mentioned is opening up for athletes to fill the blacklist. A long list increases the overhead of running anonymity algorithm. For each key,  $k$ , in the JavaScript Object Notation (JSON) data, the 'n' words black list is scanned. For one key the complexity is  $O(n * k)$ . In the worst case all keys have to be iterated. In practice, if one key - value pair is masked, keys in the value is skipped, saving time. Keeping the blacklist small is key for good performance because it is scanned for every key, but will limit the privacy protection.

In our system design its much up to the athlete him selves to secure privacy taking advantage of metacode. Non-technical athletes will probably have difficulties configuring their own code and understanding this aspect of the system. A pre made set of metacodes can be made to help the non-technical customize data accesses.

Eligibility of data accesses is done with ACL, inserting participants into research projects registered in the database. It ensures only athletes with all the resources requested for the project is made participants. Upon data accesses, the project resources field in the database is scanned to see if it has eligibility to proceed.

### **7.3.4 Accessibility**

We stated in section 3.3.7 that the system should be accessible by design from many clients. The communication protocol used to transmit analytic functions to the back-end is Hypertext Transfer Protocol (HTTP), which is standard Internet protocol used for World Wide Web and used by everyone. The accessibility of the system is therefore high being able to transmit analytics from many devices. The computation is done at the back-end

and this enables light clients to use the system without any limitations. Currently, the client is only available for devices running Python as the client implementation is based on.

### 7.3.5 Scalability

Several aspects have to be considered for scalability. The first is the storage of credentials, metacode and projects. Having these available is required for the runtime to handle any RPCs. Although, its not necessary to have these details for every athlete and project. This opens up for a decentralize approach where the runtime can be deployed at many locations, scaling horizontally. The runtime can receive replicas of athletes settings and policies from a centralized unit. Machines can in addition be scaled vertical by upgrading the machine specification.

An optimization of the runtime is to take advantage of todays multiple core CPUs. Large fractions of machines are wasted when not utilizing multiple cores. As mentioned, Node runs in a single thread, but using Nodes cluster module processing can be done on all cores available.

### 7.3.6 Fault Tolerance

Failures during processing analytic function like exceptions being thrown will abort the whole request and return the error message. If the Node software crashes it is restarted with Forever<sup>7</sup> that ensures scripts run continuously. However, this only ensures availability. A system that is down one millisecond every hour has availability over 99.9999 percent, but is not reliable [44]. To be reliable the system should be able to run for long periods without interruptions.

As stated in section 3.3.3 there is no fail handling upon machine crashes or network partitioning. The system is deployed at one machine. Neither does the storage provided by MongoDB have any replication of the data. Upon hard disc errors and loss of data, athletes and projects would have to be registered again. What DigitalOcean does upon machine failures and possibility of running back-up schedules havent been researched.

---

<sup>7</sup><https://github.com/nodejitsu/forever>

### 7.3.7 General experiences

After having the system running for several months we can draw some experience and lessons learned. Our execution model works with AP writing code for doing data processing on the health data. In our case, having adopted the Remote Evaluation (REV) design from code mobility helps us provide high degree of service customization in our system by letting APs write code to provide their own needs. The opposite would have been a statically defined interface that had to be expanded every time a new need was discovered. Our approach with code approach secures a powerful and expressive interface for APs.

There is plenty of I/O accessing and writing for executions. First of all, a folder is created for the execution, then the analytics are written to the folder before execution. Then, there is the analytic functions input and output. Analytic functions use the file system to store output. A thing to notice is that the I/O overhead might have been more noticeable with a mechanic disc, because of disc seeks having to positioning the read/write head.

Debugging the data processing functions is not trivial and is time consuming. Output from STDOUT is written to a text file. Any errors from Standard Error (STDERR) will also be in this file. A solution could be a plugin for a Integrated Development Environment (IDE) like RStudio<sup>8</sup> where the output is piped to the built in console.

A current limitation is that the runtime only supports graphical output other than in Portable Document Format (PDF) format. A possible solution for this would be to read execution folder before deleting it, and return images in an archive (such as ZIP) in the HTTP response. This havent been a big issue as both the R and Python supports plots in PDF. PDF is also vector based, making it possible to scale the plot without losing quality.

---

<sup>8</sup><https://www.rstudio.com/>





# Chapter 8

## Concluding Remarks

This chapter presents our achievements, gives some concluding remarks and outlines possible future work.

### 8.1 Achievements

In this project, we have designed, develop and evaluated a system for doing analytic in R and Python, on Fitbit and RunKeeper data archives. The problem definition is as follow:

*This thesis will explore the problem of connecting statistical analytic tools like R, Matlab and Excel to personal body sensor data archives like FitBit, RunKeeper, Python and the ZXY Sport tracking system. A language binding and runtime will need to be constructed. The runtime will be dynamically extensible by user-code to give users high control of what data that is accessible. Possibility for privacy preserving will be taken into concern in the system design. Several analytic functions will be developed to demonstrate the system and end-to-end latencies will be measured*

In the requirements specification we stated the requirements and outlined an abstract system architecture. A goal for our design was to provide athletes with high customizability and operative consent to deal with changing environments.

In design chapter, we outlined the overall system design of the system after reviewing and reflecting over several system designs including client-Web Service architecture and client-proxy-Web Service architecture. We landed

on a mobile code approach following Remote Evaluation (REV) paradigm where code is being transmitted from clients to the runtime and executed there. Further in the design, we introduced how the requirement specification was transmitted into the system design focusing on providing extensibility, eligibility and privacy preserving through metacode, Access Control List (ACL) and anonymizing.

In the implementation chapter we looked at the details. We decided to use JSON-RPC as the communication between statistical tools and the runtime for its ease of use and few requirements attached.

Then we presented use-cases showing the system in action. Several analytic functions written in R using Fitbit data was shown analysing activity and sleep patterns. A simple analytic function written in Python using RunKeeper data was given as proof of extensibility. Lastly a demonstration of the website was given.

In the evaluation a discussion if constraints and metrics set for the non-functional requirements was achieved. For extensibility, we have achieved to dynamically extend the runtime with user provided code through the metacode concept. Also in context of extensibility, the system proves to be extensible to new statistical tools and data archives by providing support for Python, RunKeeper, R and Python. For privacy preserving ACL were introduced for fine-grained access control and a novel anonymizer algorithm.

Performance measuring was done through experiments focusing on latency as the most important metric. This showed that the end-to-end latency was over 20 seconds when a relatively small data set was requested. An optimization, taking advantage of parallelism in the runtime, showed that the end-to-end latency could be reasonable when RPCs are not processed sequential. Currently, using Fitbit and RunKeeper as data archives for analytics proved to have some constraints in concern of latency and rate limits. With smart caching/storing of data and preemptive crawling the web services can become useful data sources for professional sport clubs to integrate with.

## 8.2 Conclusion

We have achieved to build a dynamically extensible system for executing data analytic code written in R and Python, allowing the tools to use data from

Fitbit and RunKeeper. The system successfully encapsulates the burdens of privacy concerns and authentication for interacting with these kinds of services. The system is easily extensible to new statistical tools and data archives. User provided code extensibility of the system through metacode where athletes can control data accesses and check for privacy violations is the biggest contribution of this thesis.

## 8.3 Future Work

Privacy and security have to be taken into concern. As suggested in the evaluation, a sandbox might be the solution to restrict executing code to gain access to the file system and to use all the resources (CPU, memory) of the runtime. In addition functions chained can access output generated by previous functions in the chain. How to restrict functions in chain in concern to secrecy and integrity haven't been thought true and need to be worked on.

In the design system we discussed some benefits with REV. One of them was push-based result of executions. Analytics can be run when data is updated, pushing the result to clients. This would improve the usability of the system. Clients get result on the fly when there are updates without requiring any effort.

The client code has now a hard coded username and secret key embedded. In addition, if more files are embedded, they are signed with the same secret key. Sharing analytic function is for athletes giving away access to use their data, passing on privilege. A method for including a analytic function in the request that is already signed is required. Another approach would be to save the code at the back-end and provide the Hash-based Message Authentication Code (HMAC) instead.

Another improvement that needs to be added is supporting multiple outputs. Now only PDFs are supported. Statistical tools may generate PNG images or files. This would not require much effort for a simple solution scanning the root folder of the execution, read the files and send them to the client.

Lastly, a way to send a context object with the analytic function(s) and have it available during the execution can be useful for analytics doing machine learning were computations builds on previous results.



# References

- [1] Json-rpc. <http://json-rpc.org/>.
- [2] More pain but no gain: More americans are exercising but poor diet means they are not losing weight. [dailymail.co.uk](http://dailymail.co.uk).
- [3] Connected continuous care. [Microsoft.com](http://Microsoft.com), March 2011.
- [4] Giovanni Vigna Alfonso Fuggetta, Gian Pietro Picco. Understanding code mobility. *IEEE Transactions on software engineering*, vol 24, MAY 1998.
- [5] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentzner. Macarons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Network and Distributed System Security Symposium*, 2014.
- [6] Fabian Breg, Shridhar Diwan, Juan Villacis, and Jayashree Balasubramanian. A distributed object model for the java system. *Proceedings of the USENIX 1996 Conference on Object-Oriented Technologies Toronto, Ontario, Canada, June 1996*.
- [7] L. Brenna and D. Johansen. Configuring push-based web services. In *Next Generation Web Services Practices, 2005. NWeSP 2005. International Conference on*, pages 6 pp.–, Aug 2005.
- [8] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Commun. ACM*, 32(1):9–23, January 1989.
- [9] European Commission. Attitudes on data protection and electronic identity in the european union. *Special Eurobarometer 359*, 2011.
- [10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*, 2004.

- [11] Peter Dizikes. Sports analytics: a real game-changer. *MIT News*, Mar 2013.
- [12] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002.
- [13] John Fox and Robert Andersen. Using the r statistical computing environment to teach social statistics courses. Department of Sociology, McMaster University, Jan 2005.
- [14] Robert S. Gray. Agent tcl: A transportable agent system. Department of Computer Science Dartmouth College, Department of Computer Science, Nov 1995.
- [15] Håvard D. Johansen Svein Arne Pettersen Pål Haloversen and Dag Johansen. Combining video and player telemetry for evidence-based decisions in soccer. *Regional Centre for Sport, Exercise and Health - North*, 2013.
- [16] Jonathan Howard. Sports science and big data opportunities, oct 2013.
- [17] Fitbit Inc. Fitbit api, <http://dev.fitbit.com/>.
- [18] FitnessKeper Inc. Health graph api. [healthgraph.com](http://healthgraph.com).
- [19] Joey Jablonski. Introduction to hadoop. *A Dell Technical White Paper*, 2011.
- [20] D. Johansen and J. Hurley. Overlay cloud networking through meta-code. In *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, pages 273–278, July 2011.
- [21] D. Johansen, M. Stenhaug, R.B.A. Hansen, A. Christensen, and P.-M. Hogmo. Muithu: Smaller footprint, potentially larger imprint. pages 205–214, 2012.
- [22] Dag Johansen, Håvard Johansen, and Robbert van Renesse. Environment mobility: Moving the desktop around. In *Proceedings of the 2Nd Workshop on Middleware for Pervasive and Ad-hoc Computing, MPAC '04*, pages 150–154, New York, NY, USA, 2004. ACM.
- [23] Dag Johansen, Keith Marzullo, and Kåre J Lauvset. An approach towards an agent computing environment. *Tekniske rapporter / Institutt for informatikk 33(1998)*, 1998.

- [24] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating system support for mobile agent. *Proceedings of the 5th. IEEE Workshop on Hot Topics in Operating Systems, Orcas Island, Wa, USA (4th-5th May, 1995)*, 1995.
- [25] Håvard D. Johansen, Wei Zhang, Joseph Hurley, and Dag Johansen. Management of body-sensor data in sports analytic with operative consent. In *of the 2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*. IEEE, April 2014.
- [26] Dogulas Laney. 3d data management: Controlling data volume, velocity and variety. Gartner, Feb 2001.
- [27] Dana Liebelson. Are fitbit, nike and garmin planning to sell your personal fitness data? motherjones.com, Jan 2014.
- [28] May. Strava, popular with cyclists and runners, wants to sell its data to urban planners. wsj.com, 2014.
- [29] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1997.
- [30] Robert A. Muenchen. The popularity of data analysis software. r5stats.com.
- [31] Min Mun, Shuai Hao, Nilesh Mishra, Katie Shilton, Jeff Burke, Deborah Estrin, Mark Hansen, and Ramesh Govindan. Personal data vaults: A locus of control for personal data streams. In *Proceedings of the 6th International Conference, Co-NEXT '10*, pages 17:1–17:12, New York, NY, USA, 2010. ACM.
- [32] Jeroen Ooms. The rapparmor package: Enforcing security policies in r using dynamic sandboxing on linux. *Journal of Statistical Software* vol. 55, Nov 2013.
- [33] James O'Toole. Mobile apps overtake pc internet usage in u.s. ccn.com, Feb 2014.
- [34] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall.
- [35] Karl Rexer, Heather Allen, and Paul Gearan. Data miner survey summary. *Predictive Analytics World*, Oct 2011.
- [36] R.S. Sandhu and P. Samarati. Access control: principle and practice. *Communications Magazine, IEEE (Volume:32 , Issue: 9 )*, 2002.

- [37] Emily Singer. The measured life. *MIT Technology review*, 2011.
- [38] David Smith. R tops data mining software poll. *Java Developers Journal*, May 31, 2012., 2012.
- [39] Michael Stal. The broker architectural framework. Siemens AG, Corporate Research and Development.
- [40] James W. Stamos and David K. Gifford. Remote evaluation. *ACM Trans. Program. Lang. Syst.*, 12(4):537–564, October 1990.
- [41] Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Marius Tennøe, Espen Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Asgeir Mortensen, Ragnar Langseth, Sigurd Ljødal, Ostein Landsverk, Carsten Griwodz, Pål Halvorsen, Magnus Stenhaug, and Dag Johansen. Bagadus: An integrated real-time system for soccer analytics. *ACM Trans. Multimedia Comput. Commun. Appl.*, 10(1s):14:1–14:21, January 2014.
- [42] Melanie Swan. Sensor mania! the internet of things, wearable computing, objective metrics, and the quantified self 2.0. *Journal of Sensor and Actuator Networks*, 2012.
- [43] P. Hunt T. Lodderstedt, M. McGloin. Oauth 2.0 threat model and security considerations. Internet Engineering Task Force (IETF), Jan 2013.
- [44] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.
- [45] S. Tilkov and S. Vinoski. Node.js: Using javascript to build high-performance network programs. *Internet Computing, IEEE*, 14(6):80–83, 2010.
- [46] Chantal Tode. Mobile health app marketplace to take off, expected to reach 26 billion dollar by 2017. *mobilmarketer.com*, March 2013.
- [47] Jason Turbow. Soccer embraces big data to quantify the beautiful game, jun 2012.
- [48] S.V. Valvåg, Dag Johansen, and Åge Kvalnes. Cogset: a high performance mapreduce engine. *Concurrency and Computation: Practice and Experience*, 25(1):2–23, 2013.
- [49] Robbert van Renesse, Håvard Johansenn, Nihar Naigaonkar, and Dag Johansen. Secure abstraction with code capabilities. In *Proceedings of*



*the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, February/March 2013.

- [50] Ashlee Vance. Data analysts captivated by r's power. [nytimes.com](http://nytimes.com), January 2009.