
**Budapest University of Technology and Economics**
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

# ICT Framework for Supporting Connected Road Vehicles

*Authors*
Tamás Balogh, Tamás István Ujj

*Consultants*
Péter Ekler PhD, László Lengyel PhD

October 22, 2014

# Contents

# Abstract

The most recent issue of *The Economist* magazine's *Technology Quarterly*, an informative report on technological progress, was printed with a picture of a man driving a smartphone on its cover. Over the driver's head, a connectivity signal appears. This piece of drawing is supposed to symbolize one of humanity's newest achievements, the connected car. An article inside the report suggests, that although the number of cars with some sort of networking ability today is only 8% of the global total, by 2020 around a quarter of cars will be online. BMW is already embedding *SIM cards* in all its new cars, and by 2020, around 90% of all manufacturers' new models are likely to have them. These advancements in road vehicles, complemented with the recent emergence of mobile computing, open up yet another area of our everyday life to software services. One can imagine, for instance, how both cautious drivers and insurance companies would benefit from a service that analyzes drivers' behavior and calculates premiums accordingly. However, it is quite challenging to successfully navigate on a field, which has so many untried ideas and so little standardization, and it is also evident, that the most interesting services profit from a stable server-side infrastructure that takes their raw datasets and adds value to them. These inherent complexities urge individual service developers to unite their skills and work together, and indeed, in the recent year, a remarkable group came into existence on our campus from those, who wish to conquer this new area of software services. The cooperation involves three departments from our faculty, and several contributors from the *Faculty of Transportation Engineering and Vehicle Engineering*, reflecting the interdisciplinary nature of the field. The authors of this work have volunteered to create a *framework*, which other service developers can depend on. On the client side, features include collecting and visualizing data and communicating with the infrastructure. To aid service developers, we created an *Android* background application and multiple libraries for these purposes. On the infrastructure side, we established a counterpart that receives and makes this large amount of data accessible for processing. *Cloud technology*, being a natural companion to connected mobile devices, was a great asset in our work. Instead of investing in our own hardware infrastructure, we were able to prototype a cluster of data processing nodes using *Google*'s cloud services. On the cluster, we deployed *Hadoop*, an open-source framework for storage and processing of large datasets, and experimented with the possibility of a highly accessible data store, capable of handling *Big Data*. Based on our research in the field, we have delivered a preliminary version of the Framework, which is already used by several teams. This work summarizes our progress and discoveries we have made so far.

# Introduction

Since its birth in the mid-twentieth century, computer software has come a long way from being a smart tool for scientists and mathematicians for their calculations. After irreversibly changing how enterprises do their business and transforming telecommunications into infocommunications, we are now witnessing the process of software services changing how people live their lives. Technological innovations of the previous two decades are fusing together into concepts we could only imagine even ten years ago. Ubiquitous internet, ever more powerful mobile devices, embedded computing facilities in everyday objects and the ability to intelligently process large datasets, together make it possible to conquer another part of our life for software services. It is likely, that in less than a decade, *connected cars* (cars with internet access) will become as common as smartphones were a few years ago. Even today, ordinary road vehicles can be upgraded to connected cars with the help of drivers' mobile devices, making it possible to do the pioneering steps in the territory way before natively connected cars become ubiquitous.

In early 2014, lecturers and students from two faculties and several departments of *Budapest University of Technology and Economics* started to work out the details of a possible collaboration aimed to enter the field of the connected car. People with very different backgrounds expressed their interest in the subject, leading to a colorful group with many competences. Members of the *Faculty of Transportation Engineering and Vehicle Engineering* strengthen the project by providing their expertise regarding the vehicle domain. The *Faculty of Electrical Engineering and Informatics*, on the other hand, is responsible for the software and service relations of the project. The group unites teams from the *Department of Networked Systems and Services*, the *Department of Telecommunications and Media Informatics* and the *Department of Automation and Applied Informatics*. All teams have ideas for connected car services, and if most of those ideas become reality, the group will be able to present a whole ecosystem of related services. The authors of this paper have created a framework, to help making all this happen.

The fact, that this project has come into existence, showcases the ability of our campus to provide an environment in which engineers and computer scientists from different faculties and departments can put their skills together in order to come up with a large-scale practical solution to a highly relevant contemporary problem. Our industrial partners are already expressing their interest and curiosity towards the concepts. The project will hopefully also leave behind a competent team, experienced in the practical design and implementation of mobile services backed by cloud infrastructure and Big Data processing. This experience can be applied later to other domains as well, which might be the greatest value of all.

Furthermore, the project will certainly provide quality material for several *Project Laboratories*, *BSc Thesis Projects*, *Diploma Thesis Projects*, papers for conferences of *Scientific Students' Associations*, and other publications.

## Motivation and Requirements

The large number of contributing teams recruited from many students and lecturers from multiple departments of the campus made it unavoidable to design and implement a common framework that solves the problems that most service creators face. Our research provided a base for this work. In the current phase of the project, we have designed and created the *VehicleICT Framework* as is. We released a preliminary version of the Framework in September, and it is already used by service developer teams for different types of applications. In addition to that, we have also implemented some applications to demonstrate the capabilities of the Framework. These applications are the *BoardComputer*, *EngineDetails*, *SocialDriving* and *EcoDriving*. In this work, we highlight mainly the *BoardComputer*, because it demonstrates the capabilities of our Framework the best. For our convenience, we refer to the VehicleICT Framework simply as the Framework or our Framework.

The Framework is intended to provide a common ground to connected car services that members of the group are working on. By offering solutions to frequent problems, creating reusable components for common tasks and maintaining a server-side infrastructure that handles the data the services need, the Framework allows other teams to focus on their specific applications and figuring out the way they can gain value from their data. With the help of the Framework, service developer teams are able to write Android applications, which collect data from the internals of the vehicle, and send the samples to a data center. The data center is, in turn, capable of storing and processing the datasets, and most importantly, it transforms them into a highly accessible format, which makes it possible for the teams to analyze their data effectively. These analyses and summaries can provide useful information not just for the connected car services, but for the vehicle industry or smart cities as well. Furthermore, looking at the Framework from an appropriate distance reveals that it is not restricted to the special domain of connected vehicles. On the contrary, it can easily be the basis of any group of services that gain their data mostly from mobile devices and need to store and process them in large scales.

We consider our work to be a success if service developers are able to rapidly create applications on mobile clients, which connect both to the car and to our infrastructure, and the infrastructure is capable of receiving data from multiple applications, storing the datasets effectively and making them available for both batch-oriented and low-latency analytic processing. We believe that if the latter requirements are met, service developers can instantly start to realize their ideas, bringing the notion of the connected car closer to reality.

## The Concept of the Framework

As *Figure 1* shows, four distinct components of the system architecture can be identified.

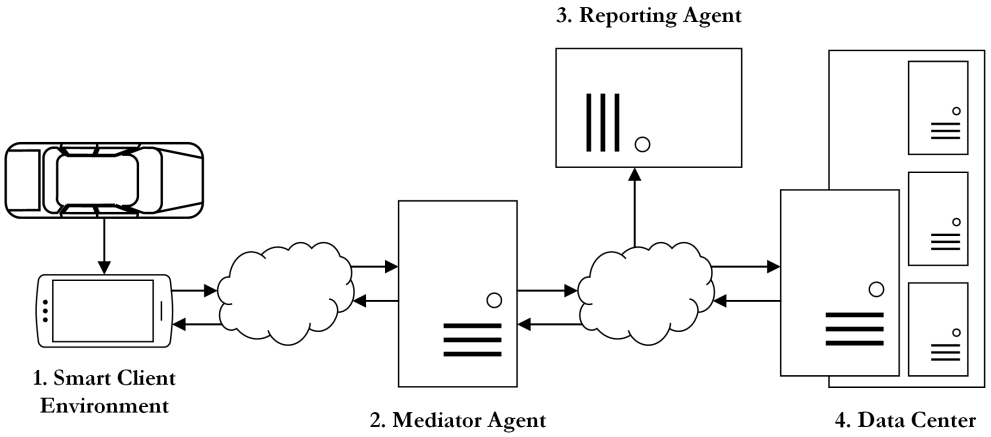


**Figure 1:** *Framework Architecture*

The mobile devices and the applications that run on them are the main interface to the service users, and accordingly, they need to be designed very carefully to meat the expectations of the users. The Framework is represented to these applications by the *Smart Client Platform*. *Figure 2* shows the main components an Android Client application interacts with.

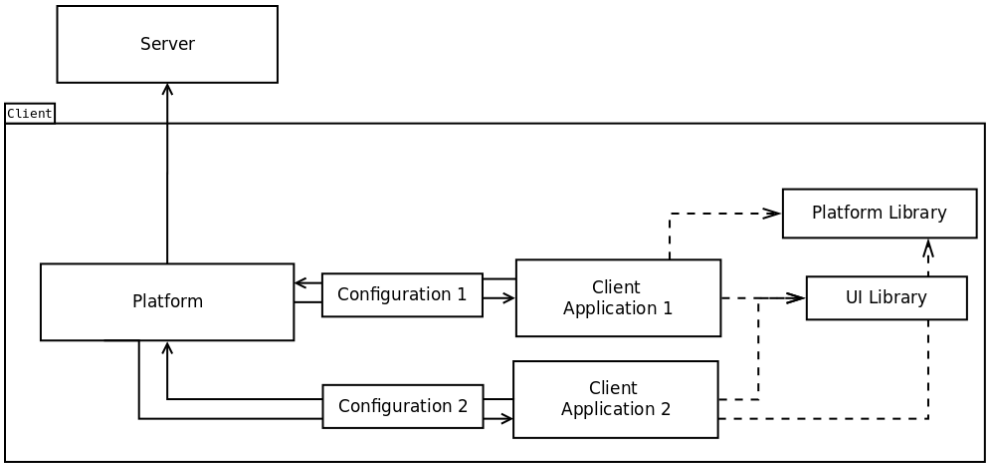


**Figure 2:** *Smart Client Environment*

*Client applications* are the central part of the Smart Client architecture. They are Android applications created by service developers, intended to use the functionality the Framework provides. Every Client application connects to the Platform, which is basically an Android application without a user interface. This component encapsulates the functionality provided by the Framework. It is responsible for data collection and communicates both with the Server and with the Client applications. It functions as a bridge between the vehicle and the Client applications, and applies solutions based on vehicle industry communication standards, like OBD-II or CAN bus. Before connecting to the Platform, every Client application defines a Configuration, which is used to influence the data sent to the Server, for example by forbidding sending location information, or by specifying the frequency of the data sampling. In order to create a Client application, developers need to include the Framework libraries in their projects. The Platform library contains the client-side implementation of the Platform, and the UI Library

provides useful UI widgets and view containers for uniform user interface development.

The *Mediator Agent*, as its name suggests, plays a central role in the Framework architecture, functioning as a mediator between the mobile devices and the Data Center, and providing APIs that other components can use, like uploading and querying the data. It also maintains operational data for the services, most importantly the *Data Contract*, which is an agreement between the server and the client regarding the content and format of the datasets. The Contract is also used to automatically generate the environment for the services in the Data Center.

The *Reporting Agent* encompasses business intelligence tools that can be used to analyze the data in an offline way. Although it is not the subject of this paper, its role is briefly described in *Chapter 3*.

The *Data Center* also plays a crucial role by being the workhorse of the infrastructure. Built on the *Apache Hadoop* framework [19], it stores the data of the services in a distributed manner, and order them into analytic data stores, making it possible to do distributed processing on them, like data mining or machine learning algorithms. Almost real-time stream processing of incoming datasets, bypassing the data store, is also possible.

*Figure 3* shows the components the Framework provides for every service, leaving out the Reporting Agent, which connects to the Data Store directly.

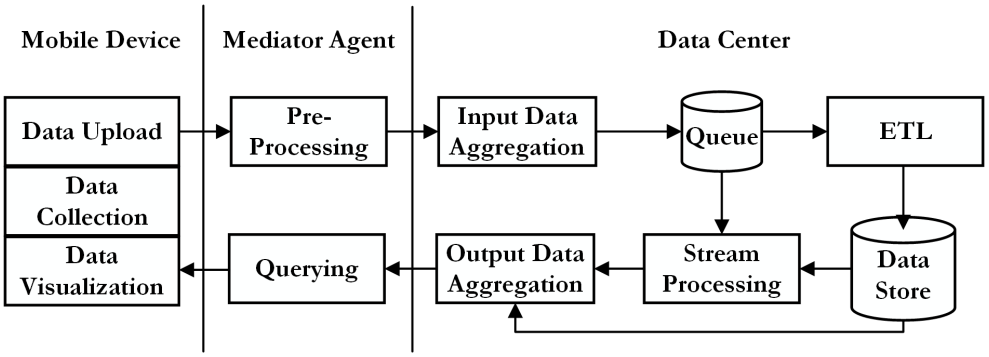


**Figure 3:** *Framework Components of a Service*

## The Structure of the Paper

The paper is divided into three sections. *Chapter 1–2* describe the Android-related aspects of the Framework, *Chapter 3–5* discuss the backend infrastructure, and *Chapter 6–7* present a case study and list related works.

**Chapter 1 (Smart Client Platform for Connected Cars)** starts with the client-side component of the Framework, describing the vehicle-related data collection and the scheduling of the sampling. Later on, additional features, such as configuration and multi-application support are presented. The chapter also discusses features that aid debugging, and shows how service developers can utilize the client libraries.

**Chapter 2 (Communicating with the Infrastructure)** presents the details of communication between the clients and the infrastructure. The chapter starts with the common data model, then the discussion of pre-processing and serialization follows. The last section of this chapter describes the communication endpoints clients use to connect to the infrastructure.

**Chapter 3 (Data Services)** introduces the services the Framework provides to the Smart Clients on the Infrastructure side. It discusses in detail, what the responsibility of a service developer is if he or she wants to use the Framework, and presents the architecture of the Mediator Agent. After that, a quick summary of the Reporting Agent follows. Finally, scaling up options are examined, focusing on moving into the Cloud.

**Chapter 4 (Data Processing Infrastructure)** starts by estimating the storage needs of a typical connected car service, which demonstrates the need for a *Big Data* class infrastructure. After this demonstration, an introduction to the Apache Hadoop framework follows, which describes the architecture on a high level and presents the Hadoop Ecosystem.

**Chapter 5 (The Data Center)** describes first, how we prototyped a Hadoop cluster with limited resources in the Cloud, examining how this solution affects the performance of the Framework. Given that, the soul of the Data Center is presented in the form of the main Data Store where vehicle data is accessible for high-performance querying. The discussion follows the dataflow, starting with data ingestion, and following through the transformation and processing steps. The chapter ends with a benchmark on the performance of our Framework, focusing on query performance.

**Chapter 6 (Case Study)** presents an already implemented mobile application using our Framework. This chapter demonstrates the client-side usage of the Framework with the help of the *BoardComputer* application.

**Chapter 7 (Related Works)** investigates related works in the connected car domain, and compares our Framework to them.

## Acknowledgments

We have received and continue to receive many help from other members of the team, as well. Péter Ekler and László Lengyel coordinate the work of the students. Dániel Zolnai is maintaining the reporting infrastructure that builds on the Framework, and is examining possible components for stream processing of incoming datasets. Tamás Agócs is developing uniform user interface elements for the Android applications. Attila Szőlősi is creating web-based tools to aid the communication between developers of the Framework and developers of the services that are built on top of the Framework. Patrik Mihályfi and Tamás Szabó are both working on custom devices, which make it possible to connect to the internals of the vehicle from mobile devices. Additionally, we also receive helpful feedback and many ideas for improvements from our 15–20 partners, who are developers of connected car services, accordingly the users and stakeholders of our Framework.

# Chapter 1

# Smart Client Platform for Connected Cars

Mobile devices have a crucial role in our proposed Framework. The Smart Client side is divided into two major parts. The *Platform* collects data samples from the internals of the vehicle and from their environment, and forward them to a smart infrastructure, so in the entire concept, this component plays the role of the data source. The *User Interface Library* is responsible for displaying vehicle-related information. The service developers provide the third component, which is the Client application that is built on the Framework. In this chapter, we describe the Smart Client Platform, explain its features, and depict the way it serves Client applications.

## 1.1 Goals and Design

During the design of the Platform, our main goal was to provide a set of real-time vehicle information for Android application developers, and to supply these data to the backend for further analysis. To satisfy these needs, the Platform should meet the following requirements:

- connecting to vehicles and sampling vehicle data,
- pre-processing raw samples,
- sending pre-processed information to the infrastructure,
- supplying Client applications with the data,
- providing information from different sources transparently to Client applications,
- allowing application developers to set up custom behavior,
- serving multiple connected applications.

In order to connect to the vehicles, the Platform creates a Bluetooth [4] connection between the Android device and the car sensors, using communication methods that are standard in the vehicle industry. The raw samples, available through this connection, should be converted into a developer-friendly form during pre-processing, and than the information has to be forwarded to the Clients and to the infrastructure. To provide these functionalities in parallel, the Platform should use independent threads and scheduling mechanisms. To support several connected applications without communication overhead, the Platform itself must be a *Singleton* application, providing connection interfaces to the Client applications, and must serve the applications in parallel and independent of each other. Each Client application must be able to configure the Platform to satisfy its needs, and the Platform has to operate according to these configurations. The Platform should handle different sources of information (vehicle sensors, mobile device sensors) and provide them to the Client applications as if they were coming from a single source.

## 1.2 Data Sources

Based on the source of the information, vehicle-related data can be divided into two groups: data from the internals of the vehicle and data from the Smart Client. The source of vehicle data are the sensors in the connected car (e.g. Fuel pressure sensor), and the source of Smart Client data are the sensors in the Android device (e.g. GPS data). Although the Platform acquires these data from different sources, this is transparent to the Clients.

### 1.2.1 The Vehicle as a Data Source

In a vehicle-related framework, it is inevitable to have some kind of connection between the Smart Client and the vehicle. One way to connect to the car is a special interface called *On-Board Diagnostics*, but it is also possible to use the *CAN bus* of the vehicle.

*On-Board Diagnostics* is a vehicle industry standard, serial debugging interface, designed by car manufacturers for testing and maintenance purposes. The development of early On-Board Diagnostics systems was initiated by Volkswagen in 1969, and later on, many manufacturers started to work on their own implementations. This lead to different manufacturer-specific systems, and to the need of standardization. We used On-Board Diagnostics II [3] in our solution, because it is the most widespread. OBD II is a mandatory part of the cars sold since 1996 in the USA, and EOBD (European version of OBD II) [40] is required in every car manufactured since 2003 in the European Union. The implementation of this standard uses a special interface that can be found under the driver panel in vehicles. Although this interface is wired and serial, fortunately Bluetooth adapters are available to convert the serial communication into a wireless, Bluetooth-based solution that mobile devices can connect to. Using this solution, every Bluetooth adapter must be paired with the mobile device, and later string based queries and responses can be interchanged over this connection.

**Figure 1.1:** *OBD II Bluetooth Adapter*

On-Board Diagnostics standard specifies a number of *Parameter Identifiers (PID)* [3], to access the debug information provided by different sensors of the car. Every data field has its own *PID*, and to access the value of the field, this *PID* must be sent as a query parameter. The response contains the requested *PID*, and the value of the data field in a string-based, hexadecimal format. Here is an example of reading Vehicle Speed using OBD II:

```
>> 010D    (01 - Mode 1  - Data bank reading, 0D - Vehicle Speed PID)
<< 010D1F  (Code to answer (010D), value 1F = 31 km/h)
```

To send and receive OBD commands over Bluetooth, we created a communication module, based on an open-source library. We extended this library with special scheduling and multi-threading support to be able to connect it to the Platform.

Although OBD II is a widespread standard, it is worth mentioning, that there are other solutions based on *Controller Area Network (CAN) bus* [5], too. CAN bus is a multi-master serial bus for connecting *Electronic Control Units* inside the car, and it is also used by OBD II as a sub-standard. Although OBD II has its advantages, it has limitations, too. OBD II only supports a subset of sensor data available in the car, and as it hides the internal details, it is hard to extend. In order to directly access the internals of the vehicle and acquire more types of data, a specific CAN-based solution is under development. This solution consists of an Android-side implementation and a custom hardware placed in the vehicle in order to access internal information. Although this solution is still under active development, we are planning to extend the Framework to support this alternative data source.

### 1.2.2 The Mobile Device as a Data Source

In order to complement the data acquired from the vehicle, the mobile device sensors are also used as a data source. The smart devices have location sensors (GPS) and a synchronized system clock, which can improve the usability of OBD data. Location sensors provide GPS coordinates and speed information, which can be used to bind vehicle related data to a geographical location in a specific time, using the system clock. The combination of these data sources offer a lot of

opportunities for Framework users. It does not matter where the data comes from, the Framework provides it to Client applications in the same way.

## 1.3 Applications and Inter-Process Communication

The Platform is a *Singleton* Android application, with a unique architecture and a purpose to serve other clients. Unlike most Android applications, the Platform does not have a user interface, and cannot be launched via the application launcher. Only one Platform application can exist on an Android device, and only one instance of this application can run on it at once. This instance starts when the first connection attempt towards the Platform is initiated. The same instance of the Platform handles several connected applications, and as long as at least one connection is alive, the instance continues to exist. When the last connection is closed, the Platform stops. To provide this functionality, the Platform application was implemented with Android Bound Services [15].

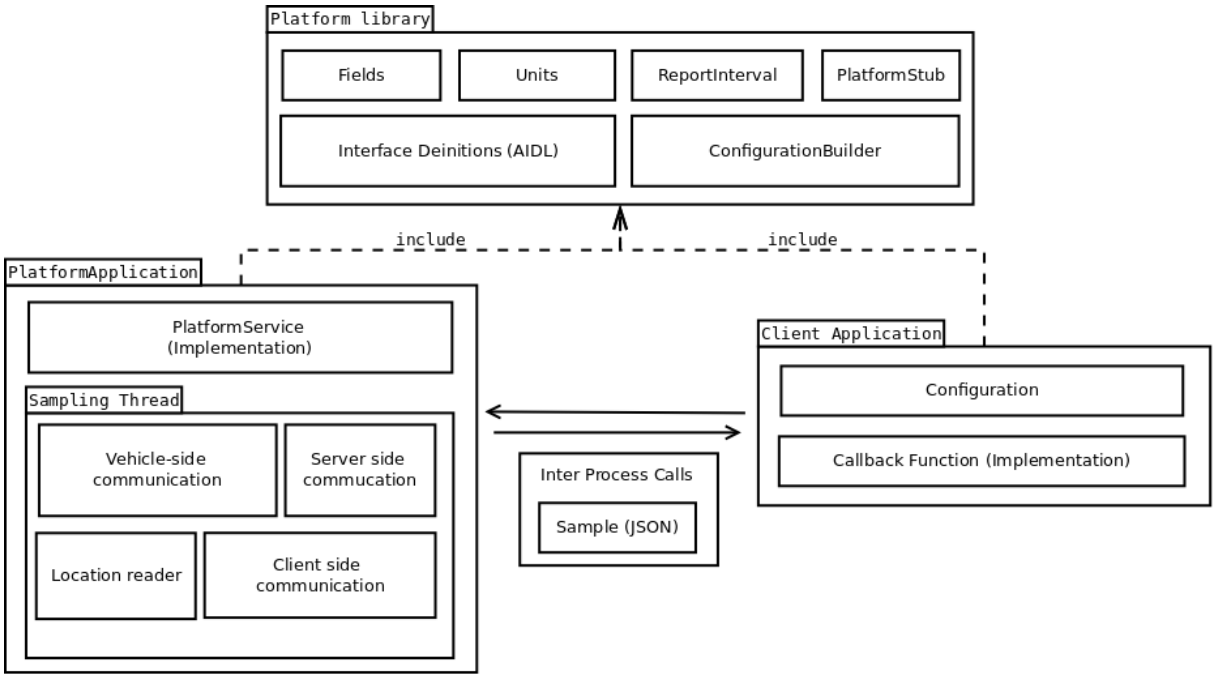


**Figure 1.2:** *Applications and Inter-Process Communication*

In order to let Client applications access the Platform service, a special communication solution must be applied. As the Platform runs as a separate application, it runs in a separate process, too. Client applications, as independent applications, run in different processes as well. To establish a connection between separate processes, the Android Interface Definition Language [14] and remote processes are used. All Client applications must declare the Platform service as a remote service in their manifest file, and the interface of remote objects must be defined using AIDL. Although the Platform is the remote service, it is not only the Platform service that has an AIDL definition. To support callback functionality for clients, the Platform Callback objects have an AIDL definition, too. These definitions are available both in the Platform and in the Client applications, but the implementations are not. The Platform Service is implemented in the

| Name | Type | Description |
|---|---|---|
| Application ID | String | Client application identifier |
| User ID | long | Client user identifier |
| Device address | String | Bluetooth adapter address |

**Table 1.1:** *Required Parameters*

Platform application and the callbacks are implemented in each Client application separately. During inter-process communication, every object must be serialized, so all non-primitive data are stored in a string representation (in a JSON format). The detailed structure of components can be found in *Figure 1.2*.

## 1.4 Configuration

Although only one instance of the Platform runs per device, more Client applications can use the same Platform. As Client applications require different data in different ways, they need to use the Platform with different preferences. To provide this, all Client applications must define a *Configuration* upon connecting to the Platform. Configurations consist of required and optional preferences. *Required* parameters must be defined by every application to build up a connection, but *optional* preferences have a default value, which can be modified if needed.

### 1.4.1 Required Parameters

Configurations have three required parameters that all client applications must supply. Upon connecting to the Platform, every application must set an *Application ID* in a string format. It must be a unique value, because it identifies the application. This *ID* is used by both the Platform (for callbacks, and scheduling) and the server (to separate information for different applications).

The second parameter is *User ID*, which is a *long*, and identifies a Client application user. This information is used by the server for user-related reports. Although the Framework supports user handling, the authentication is the responsibility of the Client applications.

The third parameter is the *Bluetooth device address*. This string identifies the Bluetooth adapter that the Platform must connect to. The Platform supports only one Bluetooth device connection at one time.

| Name | Type | Default Value | Description |
|---|---|---|---|
| ReportInterval | ReportInterval | ReportInterval.Sec5 | Time between callback calls |
| RequiredFields | List<String> | All available fields | Filter for specific fields |
| ReportToServer | boolean | true | Disable server side forwarding |

**Table 1.2:** *Optional Parameters*

### 1.4.2 Optional Parameters

Optional parameters are not necessary for a connection, instead they provide and opportunity to customize the Platform behavior to fit the Clients' needs. The most frequently used parameter is *ReportInterval*. This is the time elapsed between each measurement and callback call. For the sake of more efficient scheduling, this parameter is an enumerated field, containing predefined values from 500 milliseconds to 5 hours. Each value is the multiple of all lower values (except the lowest), to ensure that at the time of a callback, all lower interval callbacks are invoked, too.

The performance of the Platform is affected the most by *RequiredFields* parameter. This parameter is used for filtering specific data fields, and consists of the list of fields which are required by the client. All non-required fields are removed from data measurement, which lowers the sampling time. The predefined name of the fields and the most common configurations are available via a common Field class available for the Clients.

With *ReportToServer*, server-side data forwarding can be disabled. This is useful when the client intends to exclude some parts of a measurement from server-side reports (e.g. disable reporting abroad), and it also aids debugging.

### 1.4.3 The Structure of the Configuration

Configurations, as all inter-process objects, are stored in a string (JSON [37]) format during communication, but they are converted into objects both in the Platform and in the Client applications. To prevent Client developers from doing string operations, and to force them to provide required fields, but keep optional parameters eligible, we created *ConfigurationBuilder*. *ConfigurationBuilder* is a class that creates Configurations, and Configurations can only be created via this object. To create this builder object, required parameters are needed and optional parameters can be set with the proper functions. After the preferences are set, a Configuration can be built with the build function. The Configuration cannot be modified later. The usage of the *ConfigurationBuilder* is shown below:

```
Configuration.Builder builder = new Configuration.Builder(appID, userID, deviceAddres);
Configuration configuration = builder.setReportInterval(ReportInterval.Sec30)
        .setRequiredFields(Fields.getAllCommands())
        .setReportToServer(false).build();
```

Before inter-process communication, this Configuration gets serialized into a string based JSON format, and later the Platform de-serializes it before usage. The Platform stores every Configurations and the callback methods connected to a Client as a *PlatformConnection* object. The Configurations control the scheduling of the sampling.

## 1.5 Multi-Application Support

During the design of the Platform, one of the requirements was the capability of multi-application support. This means that one Platform instance per device is able to serve several connected applications on the same device. In order to meet this requirement, the Platform needs connection management. When a client connects to the Platform, it supplies two special objects. These are the Configuration, and the callback methods. In the Configuration, every application sets its own identifier, which the Platform uses to identify the application, and match further requests and the callback calls to the proper client application. These two objects form a *PlatformConnection* together. Upon every connection attempt, the Platform creates this object and stores it in the connection store. Later on, each call from an application is matched to its corresponding connection object with the help of the identifier.

The connection of the applications is straightforward. Depending on the address of the connected device, a new thread is created for sampling. This thread is used during the measurement to serve all the applications, so the connected applications do not have separate threads for sampling, they are served by this single measurement thread. After the connection is established, the actual frequency, required fields, and Bluetooth device address is set, according to the configuration of the connection. Later on, no additional thread is created for the connected application. The disconnection of applications do not affect the measurement thread, the application is only removed from the list. If the list becomes empty, the sampling is stopped. As only one global thread is used for measurement, additional connected Bluetooth devices are not supported.

Until an application starts the sampling, the measurement algorithm ignores its preferences. When it starts, the settings of the sampling thread are updated. First of all, the sampling frequency is set. If the freshly started connection has lower sampling frequency needs, the frequency of the sample thread is overridden, so the lowest connected application frequency determines the operation of the sampling thread. Then the Field filtering is set, by updating the sampling thread with a new connection object containing the fields of the connection.

The Sampling thread follows a cyclic sequence. At the beginning of the sequence, the required fields are determined. Depending on the elapsed time since the last cycle, the required fields of all connections that have a valid request for that time period are placed into a field store. During the process, the field store calculates the union of the required fields at that iteration. Only the required fields are acquired sequentially from the proper data source. The list of fields is recalculated in every cycle. The global callback distributor is called using the union of required data fields.


**Figure 1.3:** *Flowchart of the Multi-Application Supporting Process*

The global callback distributor gets a Sample object containing all the data required by the application for that time period. Then the list of concerned applications for that period is calculated, and before calling the specific callback, the distributor filters the union of fields, to contain only the data the application requests. The distributor runs on the Platform service thread, independently from the measurement. Our design decision to only allow predefined time periods for sampling makes it easy to determine the active connections.

When an application stops the sampling, its required fields get removed from the sampling thread, and the application with the second lowest sampling interval will determine the sampling frequency. *Figure 1.3* shows the flowchart of the sampling process with multi-application support.

## 1.6 Showcase Capabilities

In order to let developers create and test applications without real vehicles, the Platform supports showcase mode. From the point of view of a Client, the showcase mode operates as the real-data mode. The only difference is the device address. To switch the Platform into showcase mode, a special showcase device address must be set during the configuration. The data provided during the showcase mode is a pre-recorded sequence of real vehicle information, recorded via the Platform. To provide this feature, the Platform has a scheduling solution for real vehicle data, and another for showcase data as *Figure 1.4* describes. The difference between them is the source of information. Unlike the real-data mode, showcase mode acquires data from the pre-recorded databank. This data is stored in a JSON format data file in the Platform application as an Android asset. Upon every showcase mode initialization, this file gets loaded as a data source. The further scheduling and callback invocation mechanisms are the same, which is inevitable to ensure the same behavior as in case of a real measurement. As showcase mode uses a special device address, it is not possible to use the Platform in real-data mode, and showcase mode in parallel.

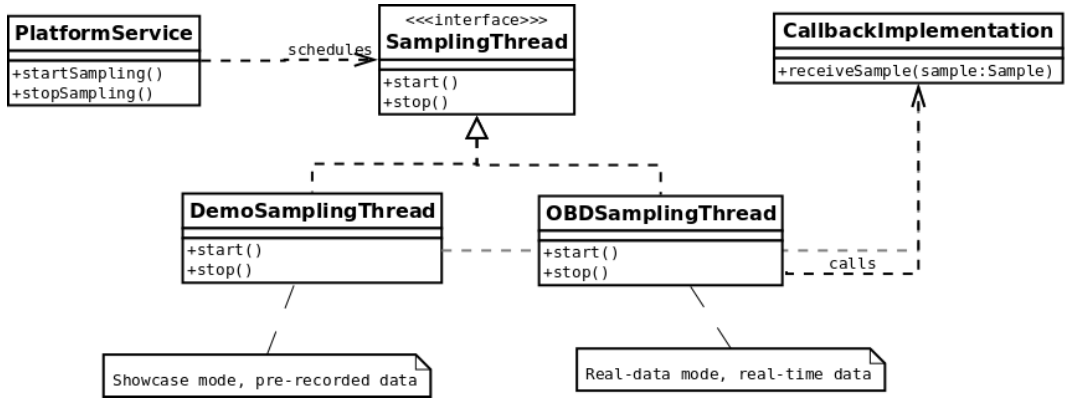


**Figure 1.4:** *The Transparent Sampling Solutions*

## 1.7 Developer Libraries

As Client applications require common software components, we created Developer libraries. Developer libraries consist of two libraries, the Platform Library and the User Interface Library. The Platform Library contains the client-side implementation of connecting to the Platform, client-side callback implementations, shared constants and wraps all the commonly used details to make the connection to the Platform easier for developers. In order to connect to the Platform, this library is required, so all the client applications must use this library. The second library is optional, and contains vehicle-related user interface elements. These consist of Android Views with reusable visibility and behavior.

# Chapter 2

# Communicating with the Infrastructure

The details of the communication between the mobile devices and the server were defined at the very beginning of the design of the Framework. Both sides need to follow these rules in order to successfully communicate, however, we designed the communication to be easily extendable. This chapter describes the structure of the communication, including conversion and serialization details.

## 2.1 The Data Model

In order to have a common structure of information, a data model has been defined. It consists of one entity called *Sample*, which contains sampled data fields as properties in a typed form. A *Sample* entity instance is the output of one step of the sampling sequence, and all of the data fields show the state of the environment at a specific time. Although the measurement process creates a sequence of *Samples*, all of them are forwarded to the server, and to the connected client applications individually. The same data model is used everywhere in the entire Platform. *Table 2.1* lists the currently available data fields.

## 2.2 Pre-Processing and Serialization

In order to convert raw sampled data into developer-friendly, easy-to-use and easy-to-transmit information, a pre-processing and serialization process is required after the sampling. The entire pre-processing, serialization and communication chain can be seen on *Figure 2.1*.

### 2.2.1 Pre-Processing

All the data acquired from the vehicle are converted from its hexadecimal string value to a decimal string format by the vehicle-side communication component (e.g. mobile device) and

| Name | Type | Unit | Source |
|---|---|---|---|
| Air Intake Temperature | int | °C | Vehicle |
| Ambient Air Temperature | long | °C | Vehicle |
| Average Fuel Economy Count | double | Miles/gallon | Vehicle |
| Barometric Pressure | long | kPA | Vehicle |
| Command Equivalence Ratio | double | | Vehicle |
| Coolant Temperature | int | °C | Vehicle |
| Engine Load | int | % | Vehicle |
| Engine RPM | long | rpm | Vehicle |
| Engine Runtime | long | s | Vehicle |
| Fuel Economy | double | l/100km | Vehicle |
| Fuel Economy Cmd. MAP | double | l/100km | Vehicle |
| Fuel Economy MAP | double | l/100km | Vehicle |
| Fuel Pressure | double | kPa | Vehicle |
| GPS Speed | double | m/s | Smart Device |
| GPS Time | long | s | Smart Device |
| Intake Manifold Pressure | int | kPa | Vehicle |
| Latitude | double | ° | Smart Device |
| Long Term Fuel Trim | int | % | Vehicle |
| Longitude | double | ° | Smart Device |
| Mass Air Flow | double | g/s | Vehicle |
| Short Term Fuel Trim | int | % | Vehicle |
| Throttle Position | int | % | Vehicle |
| Timing Advance | double | ° | Vehicle |
| Trouble Codes | String | | Vehicle |
| Vehicle Speed | int | km/h | Vehicle |

**Table 2.1:** *Fields of the Data Model*

handled as key-value pairs. In order to use these data, the string-based key-value pairs are conformed to fit into the the data model. During the process, each key is matched with the *Sample* data field, and the values are converted to the proper type.

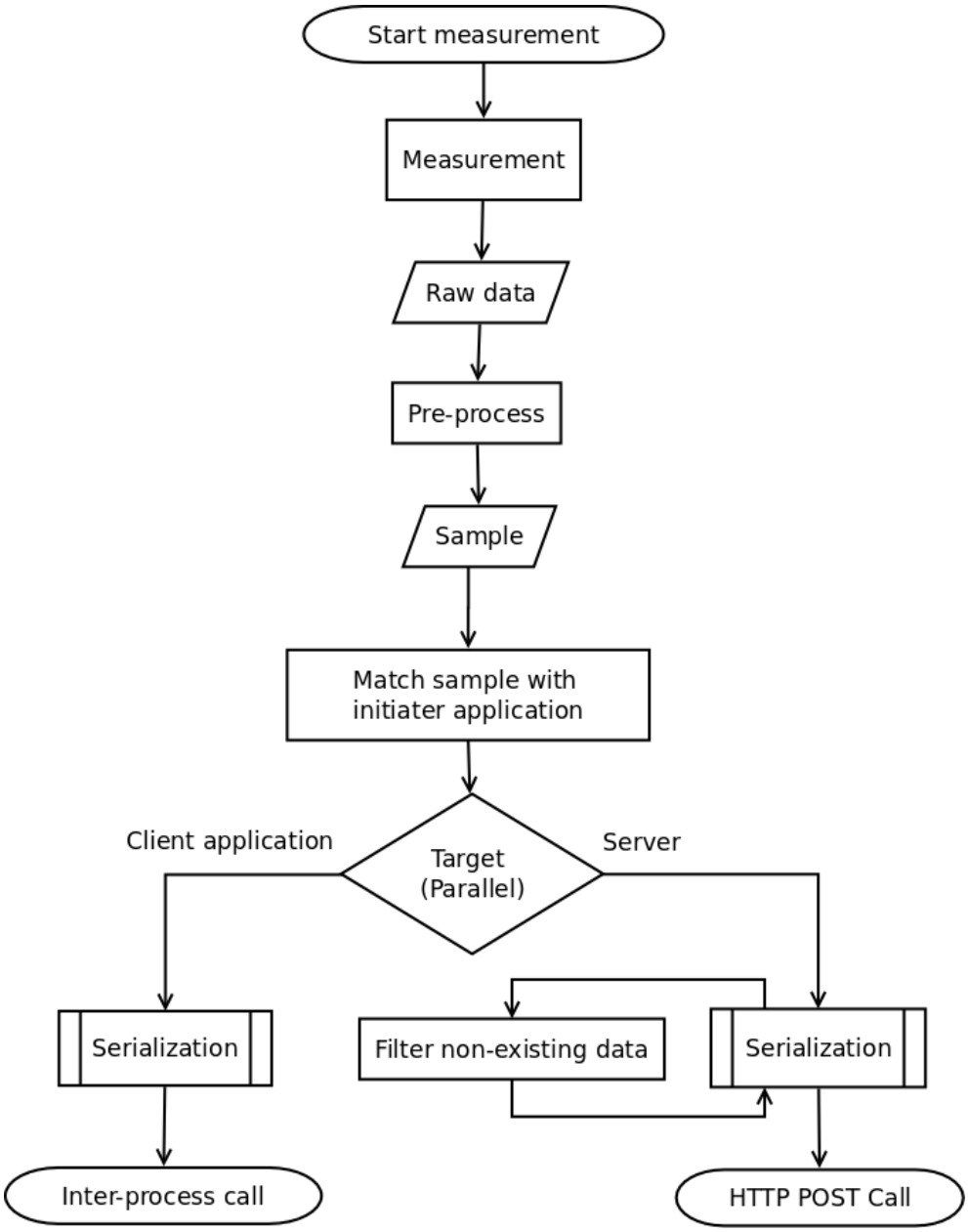


**Figure 2.1:** *Sample Processing Flow*

The most notable part of this process is the handling of unavailable data. As most of the vehicles support only a subset of the data fields, and as communication problems occur, it is vital to differentiate the unavailable data from the available ones. Zero values (and of course all values in the valid range) are not able to indicate the non-availability of the information, and primitive types cannot be *null*, so other indicator values are needed. Unfortunately, *NaN* (Not a Number) values are only supported in case of *double* values in Android (Java), but *integer* and *long* values are also used, so we used the minimal values of every type to indicate the non-availability of the data, as all the minimal values are out of the range of possible values in the domain. An

example of the process is shown below:

```
private static double extractFromStringToDouble(String dataStr) {
        ...
        if (isFilled(dataStr)) {
            String[] splits = dataStr.split(" ");
            return Double.parseDouble(splits[0]);
        } else {
            return Double.MIN_VALUE;
        }
        ...
}
```

### 2.2.2 Serialization

In order to transmit *Samples*, the fields must be converted into a marshallable, string-based, structured form. The structure of the data during transmission was chosen to be JSON [37]. JSON is a string-based data format, that resembles JavaScript objects, and besides XML [51], it is the most widespread web-transmission format today. We used Google's *GSON* library [32] to transform Java objects into JSON and back, using the *Java Reflection API* [45] to match and determine field types.

Although the *Google GSON* library is a general solution, not all the default implementations of serialization fit our requirements. In order to extend the serialization function of the library, *GSON* provides an ability to use custom serialization rules by extending its defaults. By default this library serializes all the fields, but we need to exclude non-available data fields from server side transmission, to save this unnecessary network overhead. The following lines of code show the usage of custom serializing rules in the implementation.

```
public class SampleSerializer implements JsonSerializer<Sample>
{
    @Override
    public JsonElement serialize(Sample obj, Type type, JsonSerializationContext jsc) {
        ...
        if (obj.getShortTermFuelTrim() == Integer.MIN_VALUE) {
            jObj.remove("Short Term Fuel Trim");
        }
        ...
     }
}
```

## 2.3 Communication

In order to connect to the server, a HTTP-based REST [49] communication solution is used. The server accepts a single *Sample* in a serialized (JSON) format, on an application dependent URL, as a HTTP Post call. After serialization, the Platform assembles the proper Post call, containing the *Sample* as the query body, and then sends it to the specified URL, using the Wi-Fi or mobile internet connection of the mobile device.

# Chapter 3

# Data Services

Having presented the *Smart Client component* of the Framework, in this chapter we concentrate on the middle part of the architecture, the *Mediator and the Reporting agents*, leaving the discussion of the *Data Center* to the following chapters. The *Mediator Agent* functions as an intermediary between the clients and the Data Center, and also handles the administration of the services that are built on top of the Framework, so it is a crucial part of the system, and the main subject of this chapter. The *Reporting Agent*, on the other hand, is an auxiliary component of the Framework. It connects directly to the Data Center, considering it effectively a very large analytic database, and provides tools to create reports and summaries. Finally, we examine the possibility of moving the Mediator Agent into the cloud for scalability reasons, building on top of a *PaaS* (Platform as a Service) [52] architecture.

## 3.1 The Mediator Agent

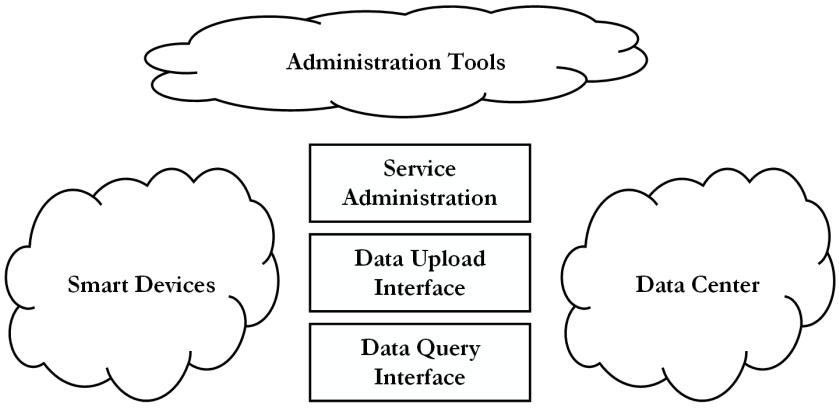


**Figure 3.1:** *Roles of the Mediator Agent*

The Mediator Agent is a server component with three distinct roles. First, it provides endpoints to the mobile clients where they can send their data, insulating them from the details of the Data Center. This way the data aggregation on the edge of the Data Center can be changed independently of the mobile applications and vice versa. Secondly, it makes communication possible in

the other direction by providing a query interface to the data of the service. Furthermore, the agent is responsible for maintaining the operational data of the supported services, like *Data Contracts*, and providing administrative functions to service developers.

### 3.1.1 Service Administration

To create a connected car service, service developers need to figure out what data the service needs, and how samples should be processed to gain information from the datasets. The service developer should be able to influence how these things are done in the Data Center. Another consideration is that although it would be possible for the Mediator Agent to just pass the data unmodified to the Data Center, it is practical to do some pre-processing on all samples before they enter the Data Center, so the Mediator Agent needs to be familiar with the structure of the samples. We solved this problem by requiring the service developers to enter into a contract with the data processing infrastructure. In its current form, the Data Contract is very simple, containing only information about the name and type of the samples and an alias to refer to them in the Data Center. The following Contract describes a service that collects the *Fuel Economy* characteristic of the vehicle.

```
{
        "appId" : "fuel_economy_inspector",
        "appName" : "Fuel Economy Inspector",
        "contract" : {
                "items" : [{
                                "measurementName": "Fuel Economy",
                                "measurementAlias": "fuel_eco",
                                "measurementType": "REAL"
                        }, {
                                "measurementName": "GPS Time",
                                "measurementAlias": "gps_time",
                                "measurementType": "LONG"
                        }, {
                                "measurementName": "Latitude",
                                "measurementAlias": "latitude",
                                "measurementType": "REAL"
                        }, {
                                "measurementName": "Longitude",
                                "measurementAlias": "longitude",
                                "measurementType": "REAL"
                        }]
        }
}
```

The Mediator Agent maintains these Contracts, but it is the responsibility of service developers to create them. The Mediator Agent provides API endpoints for this task, although it is very error-prone to administer the services using these low-level endpoints. That is why a web-based tool is under development, which hides these endpoints and provides an easy-to-use graphical interface to service developers. The Contracts can also be extended to contain further configuration options, mainly in connection with extra transformation options for the data processing. These extra options are not clarified right now, as the project is still in an early phase, and we

have only little experience so far about special requirements of different services. As soon as some real services are implemented, we will be able to advance in this matter.

### 3.1.2 Server Architecture

The previously mentioned roles are realized in a single server instance with the help of the Spring MVC framework [16]. Its design is highly influenced by the so-called *Ports and Adapters* pattern, alias *Hexagonal Architecture*, which is described in [9]. The architecture solves a problem very common in traditional layered applications, namely infiltration of business logic into the user interface code. Traditional layered applications have no mechanisms to detect if the layers violate the rule of separation, so it is very hard to enforce it in the long run. The main idea is, that a well-written core of the application must exist, in which no other concerns should be taken into account, but pure business logic. The core is protected so much, that any other parts of the application can only access it via a service API, even including the data layer.

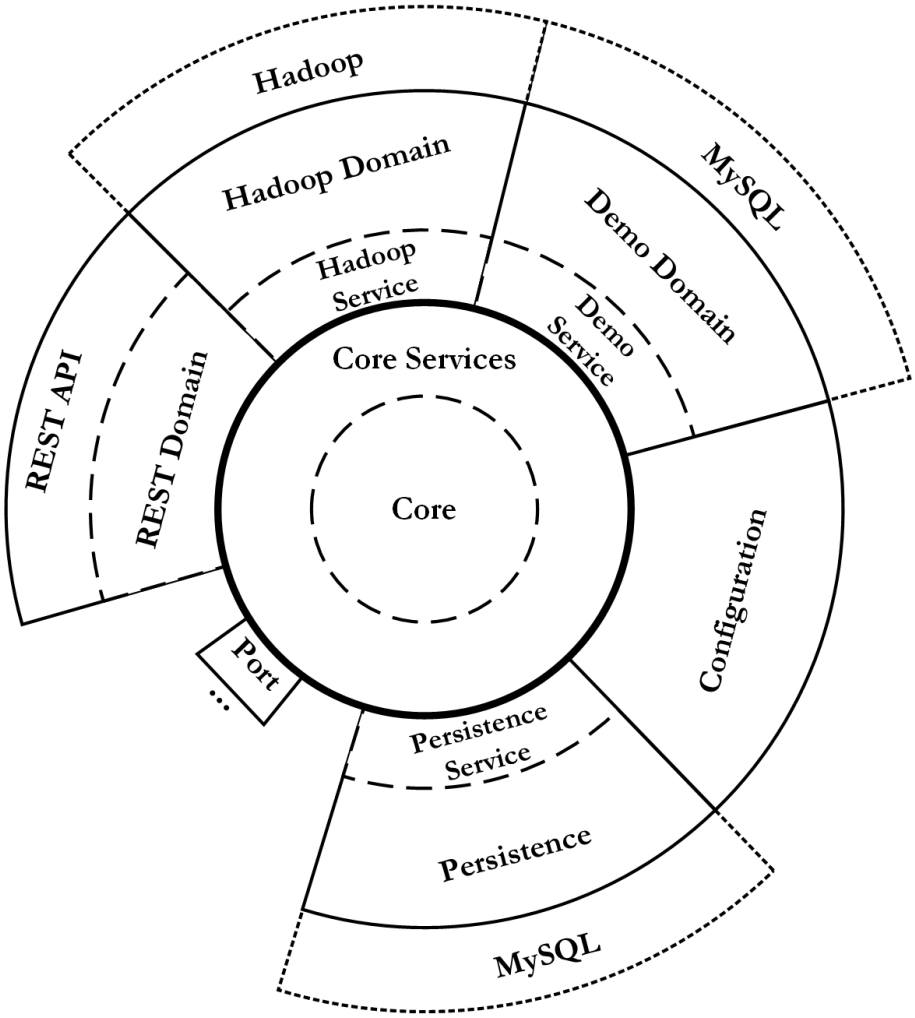


**Figure 3.2:** *Hexagonal Architecture of the Mediator Agent*

The service API provided by the core of the application can be viewed as *ports* to the internal business logic, to which other components of the application connect via *adapters*. These adapters

are also realized with their own service APIs, so components are loosely-coupled and are easily interchangeable. From the point of view of the application core, a database service is not really different from a REST interface, as both are just external dependencies to the application. That is why *Figure 3.2* represents the architecture in an unusual way, without dedicated layers.

Another advantageous consequence of the approach is that it is much easier to write tests for these loosely-coupled components, making it possible to apply advanced TDD (Test Driven Development) techniques during the development of the server [30]. Having an almost complete coverage of unit tests, acceptance tests and even integration tests allows us to perform any change on the code with great confidence, which is a crucial ability in our case, given that the Framework is used by many teams, raising unpredictable demands all the time. Thanks to the flexibility built into the design, we are able to meet these demands without destabilizing operating services.

### Components of the Architecture

To keep this paper reasonably compact, we skip all the uninteresting details that are related to the Spring MVC framework, which can be found in its documentation [16], instead we focus on the components specific to our Framework.

The *core domain* wraps concepts connected to the Mediator Agent's roles and related logic. Such concepts are a *Contract*, a *User* and a *Dataset*. The service API that encompasses these concepts in the core is based on a traditional message-driven communication pattern, in which messages containing the details of a request are passed to an interface, thus the implementation is interchangeable. The core service API is the only entry point to the system, meaning for instance that the REST domain cannot access the persistence layer directly, instead the core service API provides endpoints, which delegate requests to the persistence layer. This way the separation of concerns is completely enforceable.

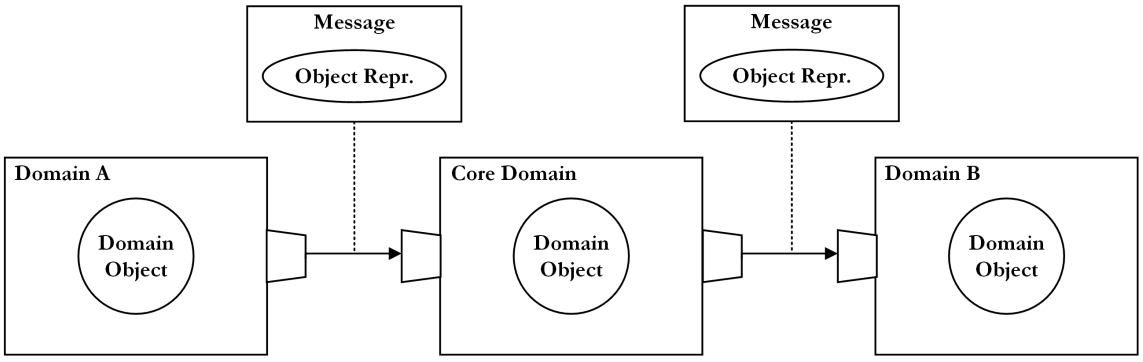


**Figure 3.3:** *Strict Separation of Domain Logic*

Enclosing the business logic into the core domain is reached by implementing domain objects for every domain separately. Instead of passing these domain object directly between different domains, a system-wide format for representing the information held by these objects exists and is used in the messages. This way domain objects strictly remain in their own domains,

eliminating any chance of leak of logic. The concept of layers addressing each other through the core domain is illustrated by *Figure 3.3*.

To store the *Contracts* persistently, a traditional persistence layer is connected to the core service API, using *MySQL* as a database engine. *Contracts*, being relatively constant, are cached in memory, because every data upload step accesses them, so it is a critical performance issue to being able to fetch them up quickly. However, this introduces a *state* to the server, which is a problem when scaling up. These concerns are discussed later in this chapter.

The *Hadoop and Demo domains* are both responsible for processing the *Datasets* and forwarding them to the Data Center. The duplication of this data processing layer has several reasons, most importantly that in the early phase of the development, the Data Center was mocked with a MySQL database. Later, when we moved the Data Center to a proper Hadoop cluster, we decided to keep this Demo layer to aid development in the future, too. We find it very fortunate, that now we have a benchmark to compare the query performance of the framework on Hadoop to, as datasets are stored in a conventional database as well as on Hadoop.

The *REST domain* is the component, which sets all the others in motion. It provides a standard REST interface, which clients can use to access the functionality of the framework. These clients include both the mobile devices and the administrative web-tools. We do not wish to go into the details of the interface, but it is worth to note that it contains endpoints for handling the *Contracts* and *Users*, and other endpoints to upload and query the *DataSets*.

### 3.1.3 The Data Upload and Query Interface

In *Chapter 2*, we described how client devices send the data samples to the infrastructure. Now we take a look at how these samples are forwarded to the Data Center by the Mediator Agent.

Interpretation of the data always happens with the help of the Data Contract. After some initial validation, data members present in the Data Contract are extracted from the sample. This is a very permissive procedure, as missing data points are handled gracefully, and extra data points, which are not specified in the Contract are simply thrown away. This way, client errors are caught by the Mediator Agent, and do not propagate to the Data Center. The extracted data points are forwarded to the Hadoop and Demo services, which arrange them to get to the next station for further processing.

Querying the datasets is very straightforward using the REST API the Mediator Agent provides. Queries are processed by the core services, and delegated to the Hadoop and Demo services, which connect directly to the data source, fetching up the requested data and handing them back to the client through the core services. Currently clients can filter on users' ID, their location and the creation time of the sample, which cover most use cases. We expect client applications to query data of their own users, or of users' close to them either in geographical location or in time. However, there are no theoretical obstacles to implementing finer grained filters, so additional filter conditions can easily be added on demand. The query engine backing this function is described in detail in *Chapter 5*.

## 3.2 The Reporting Agent

The Reporting Agent provides a managed business intelligence environment for service developers who do not wish to apply their own methods to analyze the data. It connects directly to the Data Center via ODBC [42], bypassing the Mediator Agent entirely. With the help of the Reporting Agent, users of the framework are able to create standard reports and summaries of the data, which help to discover patterns and trends. Note, that data mining, machine learning and other intelligent data processing algorithms should be implemented in the Data Center and not in the Reporting Agent.

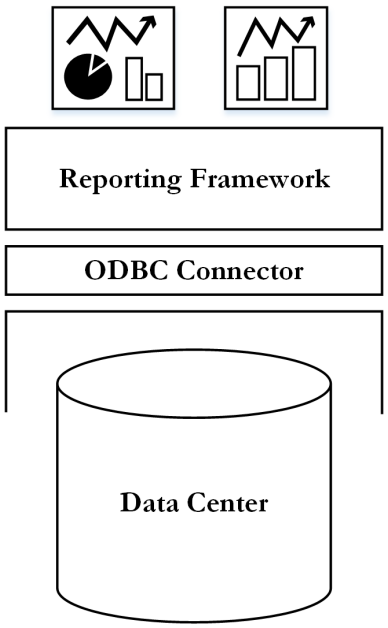


**Figure 3.4:** *Concept of the Reporting Agent*

As the Reporting Agent is not the direct responsibility of the authors of this paper, further discussion of the concept is omitted.

## 3.3 Scaling Up

Although the Framework is currently able to serve all its users, we are already considering how we would be able to scale up. The Data Center, being built on top of Hadoop, is inherently scalable, the Mobile Devices are well-distributed and the Reporting Agent has only a small number of users, so the Mediator Agent means the only bottleneck in the matter. We are not in the position to invest in a reasonably large server park, but fortunately we do not need to, as there are many *PaaS* (Platform as a Service) offerings on the cloud market today, which are built exactly for good scalability. Our Spring MVC implementation of the Mediator Agent is almost directly portable to most of these offerings, including Google's [33] and Amazon's [2], the most popular ones, though some modifications are needed. The greatest advantage of these offerings is that new instances of the server are automatically created and destroyed, reflecting the actual load of the system.

The current implementation processes requests with synchronous threads, but to fully utilize the advantages of the cloud, asynchronous processing of requests would be desirable. That can be easily reached by introducing a *queue* in which the requests arrive. Different instances of the server take the requests from this queue for processing, distributing the load among themselves. It is also not practical to store the Contracts and other related information (Users, etc.) in a fully configured RDBMS, as neither the size and structure of the data nor the predictable query patterns require it. Also, cloud offerings of RDBMSs are quite expensive. Fortunately, almost all cloud providers have some sort of NoSQL key-value or columnar data store format, which are ideal for this task. These data stores typically need to be indexed before being able to query them, for instance a dataset cannot be queried by a given column until an index is put on that column. The Contracts do not need to be searched and are always fetched using their ID, so NoSQL is a match. Switching storage method means that the Persistence Domain needs to be modified to work with the provider-specific NoSQL data store. Fortunately most of those are based on JPA (Java Persistence API) [46], so reconfiguration should not be that cumbersome.

Another problem is, that statefulness and scalability do not usually go together. Caching the Contracts in memory is not possible anymore in the cloud, because with multiple instances of the server, inconsistencies can occur. However, almost all cloud providers offer some distributed cache feature, like *MemCache* [34] for Google Cloud, which is an in-memory cache that remains consistent even with multiple server instances present.

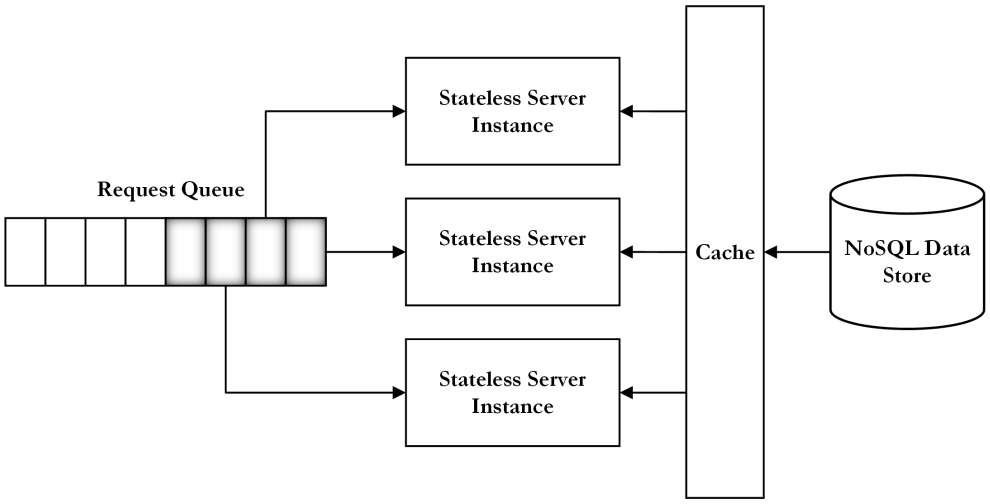


**Figure 3.5:** *Scalable Cloud Architecture for the Mediator Agent*

Moving the server into the cloud is an interesting task, and is definitely the direction for us. However, being the topic of a possible future paper, this is only a preview of our plans.

# Chapter 4

# Data Processing Infrastructure

When designing a data processing infrastructure, the first thing to do is calculating the order of magnitude of the data the system needs to store and process effectively. As the Framework needs to serve several teams and applications, choosing the right tool for these tasks influences not only performance, but also the whole architecture. In this chapter, we start by doing a rough estimate of the expected data volume. We make a point, that traditional data processing methods lack some attributes desirable for our framework, which motivated us to look for alternative solutions. One of these alternatives is the *Apache Hadoop* framework with the ecosystem that has emerged around it. Hadoop and related technologies are also described in this chapter.

## 4.1 Storage and Processing Needs

Most connected car services require mobile devices to send data to the infrastructure. The appropriate frequency of these messages significantly depends on the specific application. For instance, car manufacturers may only need some infrequent status updates about technical parameters, so they can identify flaws in the design or manufacturing of their products. On the other hand, for services analyzing driver behavior, a finer grain of vehicle data might be desirable.

For the sake of simplicity, we use a sampling interval of one second for our calculations, which covers the needs of most feasible applications.[1] It is worth noting, that a fair amount of applications are able to function with a smaller sampling frequency, but we intend to do a pessimistic estimate of necessary resources. To make calculations even simpler, we assume that a typical dataset consists of 500 bytes, which corresponds to about 20 pieces of data sent to the infrastructure in each sample. In the first phase we optimistically assume that 10% of all vehicles will be connected to the infrastructure. Again, to keep the calculations simple, we will consider only customers from Hungary. According to the HCSO (Hungarian Central Statistical Office) [44],

[1] This communication pattern has negative effects on battery usage, because mobile networks are ineffective at delivering short, periodic messages. However, we can safely assume, that mobile devices in a vehicle are connected to some sort of energy source, so we can ignore this negative effect.

there were about three million motor-cars in Hungary in 2013. Taking into account professional drivers as well as commuters who use their cars only for transport, we estimate the average time a customer would spend driving a day to be three hours.

We are now ready to calculate, how much data can be expected on a daily basis. An average customer would generate effectively 5.4 megabytes of data, leading to a 1.62 TB/day load after we project the amount to 300 thousands of drivers, which is 10% of three million. Furthermore, this is only for one service. Supporting five different services would lead to a need for resources that can handle storing and processing a data volume possibly larger than 250 TB/month, reaching into the petabyte territory in the long run.

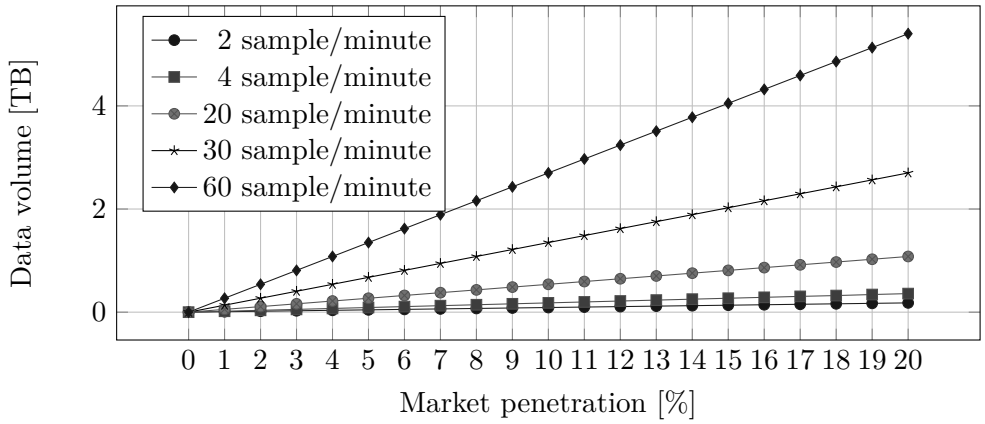


**Figure 4.1:** *Data Volume by Market Penetration and Sampling Frequency*

In the previous calculation we deliberately used inaccurate data, because it was not our intention to calculate the data volume precisely, but to grasp the order of magnitude we are dealing with. *Figure 4.1* shows, how the volume changes with the sampling frequency and the most uncertain parameters, the number of customers and the average hours a customer drives a day, expressed by their product. This product effectively shows the number of driving hours customers burden the system with each day, and we call it the market penetration. We arbitrarily consider market penetration to be 100% when all three million drivers use our service five hours a day. The previous estimate corresponds to a market penetration of 6%. According to the numbers of the figure, data volume can be kept relatively low in case of small market penetration and sampling frequency, but as soon as more customers start to use the system for sophisticated services that require large sampling frequency, the system needs to be prepared to store and process many terabytes of data day by day. It is also clear from looking at the figure, that data volume grows linearly with market penetration, which imposes an expectation on the system to scale linearly.

## 4.2 Shortcomings of Traditional Methods

We traditionally turn to relational databases to store data for applications. Relational Database Management Systems (RDBMS) have more than four decades of history, and are excellent tools for storing and querying structured data. In this 40 years, many efforts have been made to improve the performance of these products, and SQL is widely-known and used in the industry.

These systems are deservedly the workhorses of most modern IT infrastructures, as they can handle data very effectively. Furthermore, most of them can be deployed on clusters, making them capable of handling virtually any size of data, as long as they are provided the appropriate hardware. Therefore, we cannot make the argument, that our framework and traditional databases would not be a good fit, but we make the point, that for other reasons, it is worth to consider other alternatives as well.

One critical aspect is scalability, which in our interpretation means, how much money we need to spend to store and process one additional terabyte of data. Costs of conventional parallel databases frequently involve some licensing fees and cost of the special expertise needed to maintain them, and although great progress has been made in the matter of fault tolerance, they still require above the average hardware to operate reliably. [47] This means that besides scaling out, some scaling up is also required, causing the marginal cost to grow with the data volume, which makes it difficult to reach linear scalability.

Another restriction of conventional databases is that they were invented to handle structured data. It could be advantageous in some scenarios to abandon the structure, making the load process much smoother. However, as structure is what makes conventional databases very effective at querying the data, the price needs to be paid at query time. We show later, that given the nature of our case and recent developments in other areas of the field, this price can be acceptable.

There is another negative consequence of using conventional databases, and that is tight coupling of the physical storage, the metadata layer and the query engine. Again, this is a good thing when your only concern is query performance, but if you want to use the data in more than one way, this tight coupling is very restricting. We might want to define the physical shape of the data, letting us choose the most appropriate compression and format. Applying different schemes to the same dataset could also be beneficial, considering that with this much data, occasional exploratory work is inevitable, and this is impossible using fixed metadata. Also, we definitely want to apply the appropriate tool for querying and processing the data, and although UDFs (User Defined Functions) make SQL quite universal, it might not be the best tool for all cases.

All these considerations lead us to search for an alternative solution. Eventually, we arrived to Apache Hadoop, which provides almost linear scalability, handles semi- and unstructured data well, makes it possible to separate the concerns of physical storage, metadata and processing, and is described in the following section.

## 4.3 Introduction to Apache Hadoop

*Apache Hadoop* is a framework that enables the distributed processing of large datasets across clusters of computers using simple programming models. [19] It is a platform that provides both distributed storage and computational capabilities. In its core, Hadoop is based on two technologies. *HDFS* (Hadoop Distributed File System) is a Java-based file system that provides scalable

and reliable data storage that is designed to span large clusters of commodity servers. [36] *YARN* (Yet Another Resource Negotiator) is hard to describe in one sentence, but it can be pictured as an operating system that spreads the whole cluster, allocating resources to applications.

### 4.3.1 The Hadoop Distributed File System

HDFS was originally modeled after the *Google File System* [31]. It is optimized for high throughput read and write of very large files. Accordingly, it splits up all files in unusually large blocks (64-256 MB). It is based on an architecture that resembles the classic master-slave setup. In its simplest form, it consists of a *NameNode* (the master) and several *DataNodes* (the slaves). *DataNodes* are solely responsible for storing data blocks on their local disks and reporting back their state to the *NameNode* in periodic heartbeat messages, so they have absolutely no knowledge of the nature of the data they hold. The *NameNode*, the most crucial element of the architecture, manages the cluster metadata and all the *DataNodes*. It is the NameNode that has the knowledge of where the different blocks of a file are stored in the cluster, so it needs to maintain a registry of files and blocks, and for performance reasons that is always kept in memory.

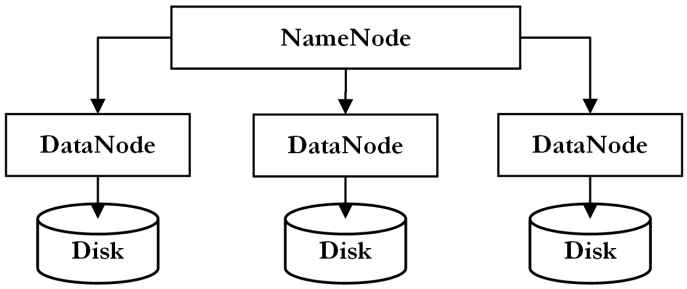


**Figure 4.2:** *HDFS Architecture*

HDFS is designed to operate on relatively cheap commodity hardware, so reliability and availability is achieved by replication of data blocks on different *DataNodes*. The *NameNode* is responsible for monitoring the number of replicas. In case of a node failure, when a replica is lost, it commands another *DataNode* to create a replica, maintaining the replication factor at all times. When reading a file, the *NameNode* can use its knowledge of cluster topology to allocate the task to the nearest *DataNodes*. The nature of the architecture also makes it very scalable, as adding another *DataNode* to the cluster is a straightforward task. In production clusters, HDFS has demonstrated scalability of up to 200 PB of storage and a single cluster of 4500 servers, supporting close to a billion files and blocks. [36]

### 4.3.2 Yet Another Resource Negotiator

One of the goals of Hadoop is that a developer should not be concerned that the application will run on a cluster instead of a single machine, and should be able to concentrate on the correctness of the data processing logic, and not the low level technical details of resource allocation. This is

in fact the exact same desire that lead to the invention of operating systems, so YARN is often called an operating system for the data center.

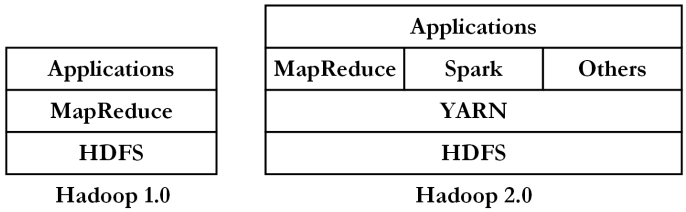


**Figure 4.3:** *Generations of Core Hadoop*

Like HDFS, YARN has a single *ResourceManager* and several *NodeManagers*, which correspond to the *NameNode* and the *DataNodes*. However, for YARN, a third type of role exists, the *ApplicationMaster*, which, as its name suggests, manages a single application, so basically it is an interface between YARN and the different application frameworks. Thanks to this interface, YARN is capable of integrating any application framework that exists for Hadoop (see *Figure 4.3*). This is a big step forward as opposed to the very recent times, when Hadoop supported only the *MapReduce* paradigm [11], which proved to be inadequate for several scenarios. With the help of YARN, applications are now able to use the framework that suits their needs the best. The *ResourceManager* is responsible for scheduling and distributing the applications among the available nodes and tracking and monitoring the scheduled tasks. *NodeManagers*, in turn, launch the application containers on the nodes and report back their status to the *ResourceManager*. One fundamental feature of YARN, which is inherited from MapReduce, is that if a failure occurs on a node, the *ResourceManager* becomes aware of the failure, and gives the task to another *NodeManager*, so eventually the job will succeed.

### 4.3.3 The Hadoop Ecosystem

Hadoop, with its foundation on HDFS and YARN, proved to be an excellent open-source framework for handling large amounts of data effectively. No wonder, that a plethora of integrated tools and frameworks emerged, leading towards what we might call a Hadoop Ecosystem. It is out of our scope to fully describe the whole Ecosystem, but we try to give a quick insight into the technologies we think the most influencing.

**Apache Flume and Apache Sqoop** [18][28] are integration tools which can be used to load data into HDFS from various sources. Flume is traditionally used for collecting logs, but it is capable of many more. Sqoop is used for the same task, but it is specialized in loading data into Hadoop from an RDBM and vice versa. We used Flume in our Framework to aggregate individual datasets in the Data Center received from the Mediator Agent.

**Apache Spark** [27] seems to be the data processing framework replacing MapReduce. [8] Indeed, it can handle all problems that MapReduce can, and many more, that it cannot. Spark is capable of running data mining, machine learning and graph analytic algorithms as well as processing streaming data. Apache Tez and Apache Storm are similar application frameworks. Spark

is also the first tool we suggest to users of our Framework who wish to do custom processing in the Data Center.

**Apache Hive and Cloudera Impala** [21][7] are both tools for querying data stored in HDFS with the use of well-known SQL queries. Hive, developed by Facebook, is a data warehouse framework that is based on the MapReduce paradigm, a legacy of earlier Hadoop versions. MapReduce is a fundamentally batch-oriented framework, excelling at some types of problems, but very lacking as a base for an interactive query engine. Since the introduction of YARN, it is possible to use other processing frameworks with Hadoop, and indeed, Hive is being rewritten right now as a community effort to be able to run on other engines, like Spark. [8] In the meantime, alternative solutions have come to life, most notably the Impala framework, which operates its own node-managers to bypass the batch-oriented processing framework, enabling interactive querying of data. Both Hive and Impala are heavily used by our Framework, as *Chapter 5* describes.

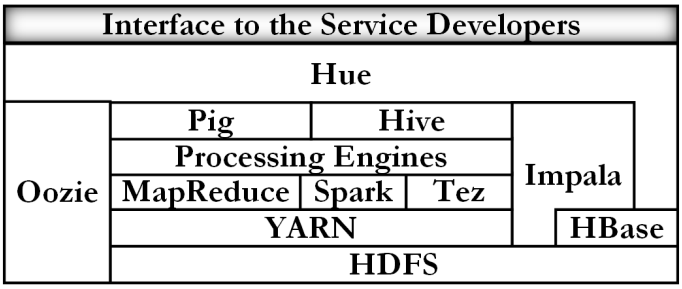


**Figure 4.4:** *The Hadoop Ecosystem*

**Apache Pig** [26] is a high level scripting language, that is traditionally used to express MapReduce-based data pipelines, and has a very short learning curve. Although it was specifically created to be compiled into MapReduce primitives, it is also being rewritten to use Spark instead [8], opening up new areas to express high-performance data transformations in a high level scripting language. Pig is the data processing tool we suggest to users of our Framework who are not familiar with Java, Scala or Python, because one of these languages is required for using Spark.

**Apache HBase** [20] is a NoSQL columnar storage, modeled after Google's BigTable [6]. It has been a part of Hadoop since the beginning, and was a great tool if short response time querying of data was needed, leaving batch-oriented jobs to MapReduce. We recommend HBase for users of our Framework who need an operational data store besides the analytic one the Framework provides by default.

**Apache Oozie and Apache Hue** [24][22] is a distributed workflow management system that is used to schedule jobs on a Hadoop cluster, as well as in our Framework. Hue is an aesthetic all-in-one GUI for Hadoop. It provides a surface on which it is possible to browse HDFS like a traditional file system, to query tables in Hive, Impala or HBase, to define and edit jobs and workflows and to schedule and monitor these jobs. We provide restricted access to Hue for the users of our Framework, being their development interface to the Data Center.

# Chapter 5

# The Data Center

In the previous chapters we described components of the Framework, which are responsible for collecting the data, but we said little so far about the way datasets are stored and made available for services. In this chapter, we finally discuss the architecture of the Data Center. During our discussion, we build on concepts presented in *Chapter 4*. We introduce the topic by demonstrating how a functioning Hadoop environment can be built with limited resources, then we discuss the Data Store, which is the backbone of the Framework. We also take a quick glance at other data processing capabilities of Hadoop, which really differentiate the Framework from traditional RDBMS-based data processing systems. The chapter ends with a measurement on the query processing performance of the Framework, which is one of the most important aspects from the service developers' point of view.

## 5.1 Prototyping a Hadoop Cluster in the Cloud

Hadoop was designed to work well on relatively inexpensive commodity hardware, applying replication on blocks of files, which provides both reliability and scalability. Traditional methods that improve reliability and performance, like RAID, which is common in server rooms, or swapping, which is present in most operating systems, are therefore unnecessary overhead to the system as a whole, so Hadoop works best when installed on plain machines without any serious improvements. Consequently, in the long run, it might be economical to own the machines of our Hadoop Cluster, even so because otherwise decommissioned servers can be readily integrated into a Hadoop cluster, since the fault tolerance of the system does not come from reliable hardware. However, if we do not already own such a server park, this can be a serious barrier to entry. Fortunately, Cloud technology made it possible to acquire a large amount of processing power for a short time, paying only for the actual usage, then break down the machines, eliminating all future costs. This way the otherwise fixed cost of necessary hardware for development becomes a variable cost, which is very advantageous for an early entry into a field. For the sole reason to make it affordable for ourselves, we chose to prototype the Hadoop cluster with the help of an *IaaS* (Infrastructure as a Service) offering [52], which helped to keep costs low.

### 5.1.1 Performance Considerations

Using an IaaS platform means, that we can request virtual machines and configure them, however we need to. Unfortunately, this freedom only lasts as long as we keep ourselves above the operating system level, because IaaS by definition hides lower level layers behind its abstraction. Those layers involve virtualization, hardware, storage and network architecture. *Figure 5.1* illustrates the stack. We can tune the operating system to meet our needs better, but not the four bottom layers. To determine if we should expect any performance loss due to building on an IaaS, we now examine these hidden layers.

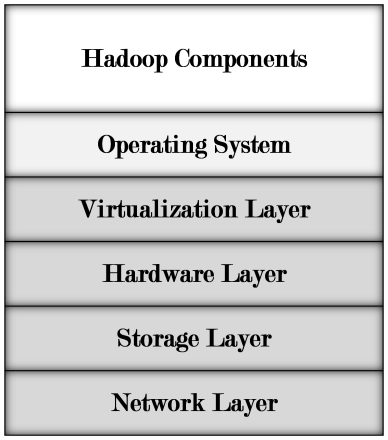


**Figure 5.1:** *Infrastructure as a Service with Hadoop on Top*

**Virtualization** imposes an overhead on the overall system, but nowadays that is hardly a big deal, and we can safely assume that Cloud providers do it better than we would ourselves. Of course this overhead would be eliminated if we used our own physical machines, but the costs would probably outweigh the gains. Anyhow, virtualization does not have any additional effects on the performance of the Framework.

**Hardware** matters very little in our case, because it does not really matter where the CPU cores or the memory come from, as long as the Cloud provider provides a reliable service for a price affordable to us. Hiding hardware details might have significant effects on our budget, but not on the performance of the Framework.

**Storage** architecture details, as we might expect, is a much more relevant issue, since one of the most important pillars of Hadoop is its distributed storage system. Performance of HDFS has a large effect on the performance of the whole Hadoop system. Unfortunately, having no say in storage methods means that we cannot eliminate features which are useful for other application areas (like RAID), but with HDFS deployed on top of them, are made redundant. Of course we can always configure our system to take these circumstances into account, for instance by decreasing the replication factor. Unfortunately, the lack of control over the storage layer may impose a significant performance loss on our Framework.

**Network** topology aids Hadoop in reaching its maximum potential. For obvious reasons, when machines are placed into different racks, the intra-rack[1] communication has a higher bandwidth

[1] Between nodes in the same rack.

than inter-rack[2] communication. HDFS itself was designed with rack-awareness, which means for example that when replicating a file block, it makes sure that one replica ends up on a different rack than the others for reliability reasons, and when reading or writing a file, the request is processed by the nearest *DataNode*, which is the one in the same rack, for availability reasons. If we have no control over, or worse, no knowledge of network topology, we are not able to utilize the rack-awareness feature of Hadoop, and that clearly means performance loss.

Deploying our Hadoop cluster on an IaaS platform has its downside from the viewpoint of Framework performance, and we need to take account of that when calculating the costs and benefits of utilizing an IaaS solution instead of operating our own servers. Until Cloud offerings which include full Hadoop functionality mature [1], it might be rational to keep our own infrastructure in the long run. However, Cloud is a great asset during development and for early operations.

### 5.1.2 Flexibility versus Stability

Cloud technologies are inherently flexible, and large Hadoop clusters benefit from stability, so this might be another point of conflict between the two approaches. Although flexibility does not necessarily mean instability, and likewise, need of stability does not always lead to refrain from change, it requires special care to make these two worlds work together.

For developing applications and trying out concepts, it is not worth it to burden ourselves with cluster maintenance problems, because we can run Hadoop on a laptop in pseudo-distributed mode with small datasets, or use some free Cloud offerings of Hadoop, which are designed exactly for that purpose. In this phase, we simply disregard stability. However, to build a Hadoop system, one cannot avoid testing on an actual functioning cluster of his or her own, so a somewhat stable development environment needs to be set up at some point during the project. Of course we would like to avoid paying for such a cluster when we do not use it. The problem is, that it is not a straightforward task to keep the state of the cluster between two development cycles, and we would like to avoid having to reconfigure it every time we restart the cluster.

Currently all Cloud providers fee for persistent storage capacity, even if not used, so if we need to have the cluster in the same state at start-up as at the last shutdown, we either need to create snapshot images of the disks and store them locally, a very cumbersome approach, or accept to pay a reduced fee even during downtime. Another difficulty is that at the time of writing, most Cloud providers disallow any manipulation of internal (network) IP addresses, so we cannot just attach the disk to a newly created virtual machine with the internal IP address it had before. There are several viable solutions to this problem. One is to use external IP addresses, which are allowed to be fixed. However, it is very undesirable for a cluster to operate based on external addresses, or even for the internal nodes to have one, because it makes it harder to maintain proper firewall rules. On top of this, communication to an external IP address counts as egress communication, even if the machine is in the same data center, and that involves extra fees. We found it more efficient to simply accept that every start-up involves configuring the nodes to

[2]Between nodes in different racks.

know about their new internal addresses, which is easily scriptable. We have written quite a few scripts to support our development method, but it is worth to note, that there are tools [23] for automatic cluster maintenance, that can work with Cloud providers, and are more suitable for production use.

### 5.1.3 Evaluation of the Concept

We have identified three phases of a Hadoop project, which are experimentation, development and production deployment. We conclude, that for production use, cloud technologies may or may not be the most fitting solution, as both technical and business parameters highly depend on the current cloud offerings. However, for the first two phases, experimentation and development, utilizing a Cloud environment to create a development and test cluster has clear benefits over operating our own machines and can be safely recommended to small teams. Our Framework would not have come to life without this concept.

## 5.2 A Hadoop Data Store

All the theoretical and technical discussions of the previous few pages were meant to provide a basis for this section, where we present how we joined the many loosely-coupled components together to form a cohesive, stable environment to store the data of the services. To be able to strongly concentrate on the Framework, we omit most details that are specific to the different services, like special transformations on the data or the exact data model, focusing instead on the aspects strictly related to the dataflow. The section is a demonstration of how components of an ecosystem, like Flume, Oozie, Hive, Impala and Spark, all designed for different purposes, can work together as a Framework.

### 5.2.1 Data Ingestion

The Mediator Agent does not perform any aggregation on the data, neither does it separate datasets for different services, so the Data Center receives all data samples for all services individually on the same entry point. However, it is not efficient to write these small samples of few kilobytes directly to HDFS, because they would eventually overload the *MasterNode*. We need to aggregate and separate these samples into service-specific collections, and *Apache Flume* is a tool perfect for this purpose. A Flume pipeline consists of a source, a channel and a sink. The *source*, which in our case receives HTTP POST messages, collects the data from external system components and places them on the channel. The *channel* can either be an in-memory or a file-based queue, depending on the reliability requirements. We used a memory-based channel, because the Framework needs performance more than reliability, as some missing samples are not as harmful as a delay for all samples. The *sink* collects the datasets from the channel and after a given time, ideally under a minute, it writes them to HDFS for every service.

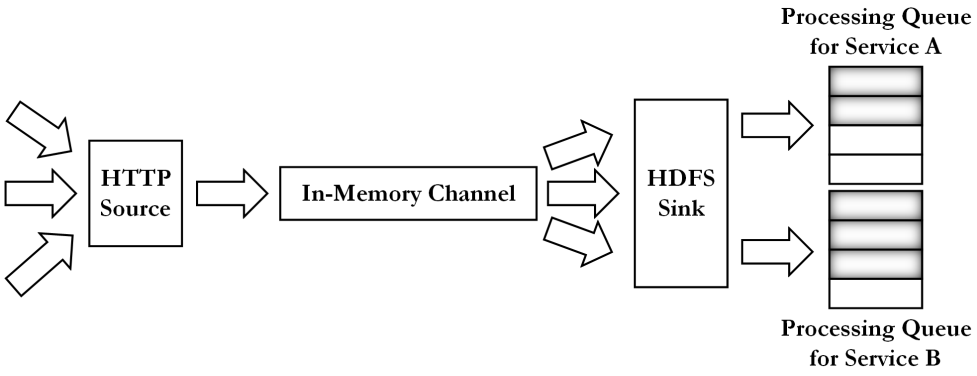


**Figure 5.2:** *Data Ingestion and Aggregation*

From the moment these aggregate files are written to HDFS, they are put into temporary queues, and are instantly available for processing. We experimented with an aggregation interval of 30 seconds, so taking network delays into account, there is a maximum of 33-35 seconds delay between the measurement and applying the processing logic. If a service needs lower delays, like streaming algorithms, with special care, data undergoing aggregation can be processed, as well.

### 5.2.2 Transformation

JSON is ideal for network communication, but it is far from an effective storage format for large amounts of data that is frequently queried. Datasets need to be transformed into a format which is compact enough, and at the same time enables low latency querying of the underlying data. *Parquet* [25], which seems to be the only wide-spread Hadoop storage format that accomplishes both expectations well, is a columnar storage format. This means it stores data points from the same column together, and not from the same row. This way it is able to apply effective compression algorithms to the data, and provide excellent analytic query performance, as analytic queries usually aggregate values from the same column. However, there is a twist: to fit standard query patterns, Parquet always stores the columnar values of a row in the same file, making it easy to fetch a single row, too.

As Parquet compresses the files significantly and provides high-throughput, low-latency query performance, we transform raw JSON data into Parquet files. This happens in scheduled time intervals with the help of the *Oozie* workflow scheduler tool, because it is not practical for previously discussed reasons to transform incoming datasets one by one. The transformation occurs every five minutes, and is performed by *Hive*, which is an excellent tool for the task. Although it lacks low latency querying capabilities, it is a very robust and fault-tolerant batch processing framework, ideal for ETL (Extract, Transform, Load) workloads. We also experimented with Pig and Impala for this purpose, but Pig has its own script language that is different from SQL, and Impala processing is very memory-intensive, furthermore it is quite unstable for long running transformations, especially compared to Hive. Nevertheless, Impala beats Hive when it comes to low-latency query performance, as our measurements show at the end of this chapter.

Converting the data into Parquet format already improves query performance a great deal, but

we go further and also partition the data store. Strong candidates for partitioning columns include users' ID, time of the measurements and the geographical location of the measurements, reflecting the most common filters used on queries. Currently partitioning by time dimension is implemented. This way services do not need to search the whole data store when looking for answers that only require recent datasets.

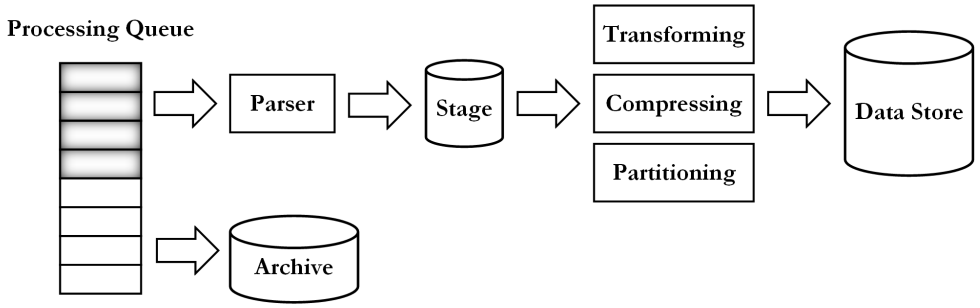


**Figure 5.3:** *Transformation Steps*

The process is essentially controlled by three script files, two of which are provided by the Framework and one is developed by the service developers. Right after raw data is taken from the service queue, the first Hive script parses the JSON samples and loads them into a staging area. The staging area is a reasonably compact representation of the recent datasets, and can be accessed either with SQL, or with other custom transformation scripts. Users of the Framework can provide their own logic at this point, which is plugged into the Framework, and is able to do custom transformations on the staged data. Finally, the second Hive script, which is provided by the Framework, transforms the dataset into Parquet format and moves it to its final partitioned location.

### 5.2.3 Custom Data Processing

The Framework provides a ready solution to services for collecting, transforming, storing and querying their data, although this might not be enough for some service developers. If a service requires special methods, one needs to build a plugin, which is also supported by our Framework.

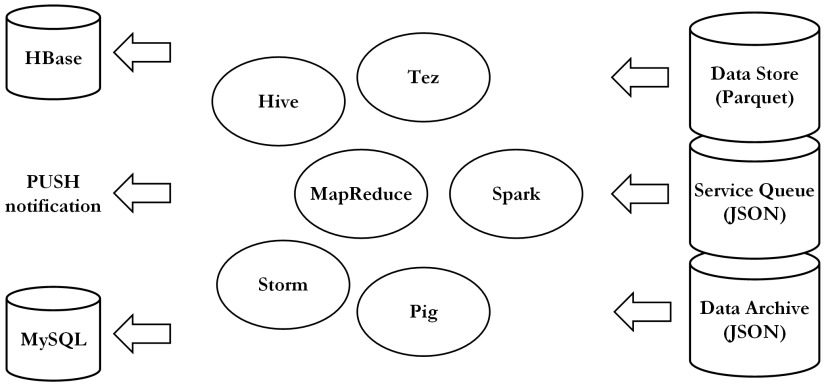


**Figure 5.4:** *Custom Processing Capabilities of the Framework*

To write custom processing logic, a service developer needs to be familiar with at least one of the many processing frameworks Hadoop has. Our preferences are for Spark, but we can

support other engines as well. However, being a special use-case and only concerning specific services, the Framework gives a free hand to service developers to implement their own plugins, although our team can advise on these matters. An example would be a service that constantly monitors the technical parameters of the vehicle and notifies the driver if he or she drives in an uneconomical way. A service like this can easily be implemented with the help of *Spark Streaming* [29] by continuously running an algorithm on raw incoming streaming data and sending a *PUSH notification* [53] to the Smart Device. We are planning to implement such a mechanism in the Mediator Agent, as it might be needed by many services. Another example for custom processing could be a service that predicts where to go if the driver wants to park the vehicle. A possible implementation of this service would periodically run a batch-job on the Data Store written in Spark or Pig and load a special table either in MySQL or in HBase for maintaining predictions for different parts of the city, which Smart Clients can query. The possibilities are virtually endless.

## 5.3 Performance of the Query Engine

As we already mentioned, Hive is not the best tool for interactive querying of the data. It might be used to create regularly generated reports, but we wanted to enable users of the data to explore the data store, drilling down and issuing ad-hoc queries. On the other hand, it is also important to make the data store accessible to the Smart Clients through the Mediator Agent, which requires low-latency query processing. That is why we decided to replace Hive at the end of the transformation flow with Impala, which working together with the Parquet file format proves to be a very effective low latency query engine. To prove that our concept really works, we designed some tests to check the actual performance of the query engine deployed on our cluster.

An algorithm was created by one of our team members, which generates data that resembles the patterns of a real car. This might have been the single most important step to move forward the evolution of our Framework, as we were able to test our concepts without having to go through with the cumbersome process of collecting data from a real vehicle. We ran a simulation of a service that sends samples to the Data Center every five seconds from each client applications. The test data runs through 20 days, although it was accumulated in a short time, corresponding to about 90-100 clients generating data all at once. It was also a test of the Framework as a whole. We tried to figure out if the Framework was capable of serving the first set of users, which consists mainly of service developers. According to the test results, we are confident that our Framework can serve several hundred users in its preliminary version already, which is quite sufficient for the initial phase of service development, although it is obvious that the Mediator Agent needs to scale out later, as we already mentioned in *Chapter 3*.

To demonstrate that Impala was a good choice, we ran the test queries on the same growing dataset both on a MySQL server and on the cluster with Hive. We tried to make the comparison as fair as possible, so four CPU cores were dedicated to all three engines. We also limited the memory for Hive and Impala processes to 8 GB, the same size that the MySQL server had

to operate on. Furthermore, we deliberately used tests that scan the full dataset, so Hive and Impala could not benefit from partitioning the data.

The first test we ran was to check how Impala scales compared to the other two. We used an incrementally growing dataset for our tests and did six measurements. The first measurement was executed on 2 million raw samples and the rest consecutively on 4.5, 6.7, 9, 11.1 and 13.2 million samples. The final dataset had a size of 10.5 GB, which the MySQL server compressed into 3.5 GB. It is impressive, that the Parquet representation of the data was only 1 GB. The test query was to calculate the average speed for every user.

*Figure 5.5* shows that Impala scales remarkably well with large datasets. Although the MySQL server performed considerably well for a few million samples, processing time quickly rose as size of the data rose, while queries issued to Impala were answered in a few seconds even with 13 million samples. Hive showed similar scaling properties as MySQL, though Hive query performance suffers from a large overhead caused by the fact, that Hive translates queries into *MapReduce* jobs. This overhead is present even in case of small datasets.

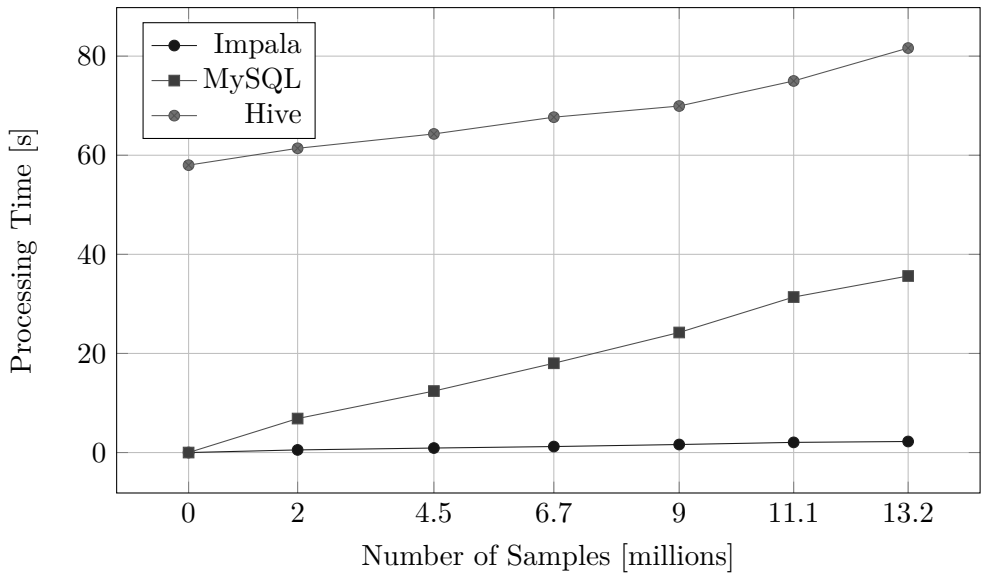


**Figure 5.5:** *Comparison of the Scaling Properties of the Engines*

We also examined how Impala performs in case of different query patterns, compared to MySQL and Hive. We used four different queries, which try different aspects of the engines.

**Full scan** counts the number of samples in the whole dataset.

**1 condition** applies a single condition on a single column.

**2 conditions** applies two conditions on two different columns.

**Aggregation** calculates the average speed of different users.

We ran the queries on the full dataset, that contains 13.2 million samples. As *Figure 5.6* shows, results are very promising.

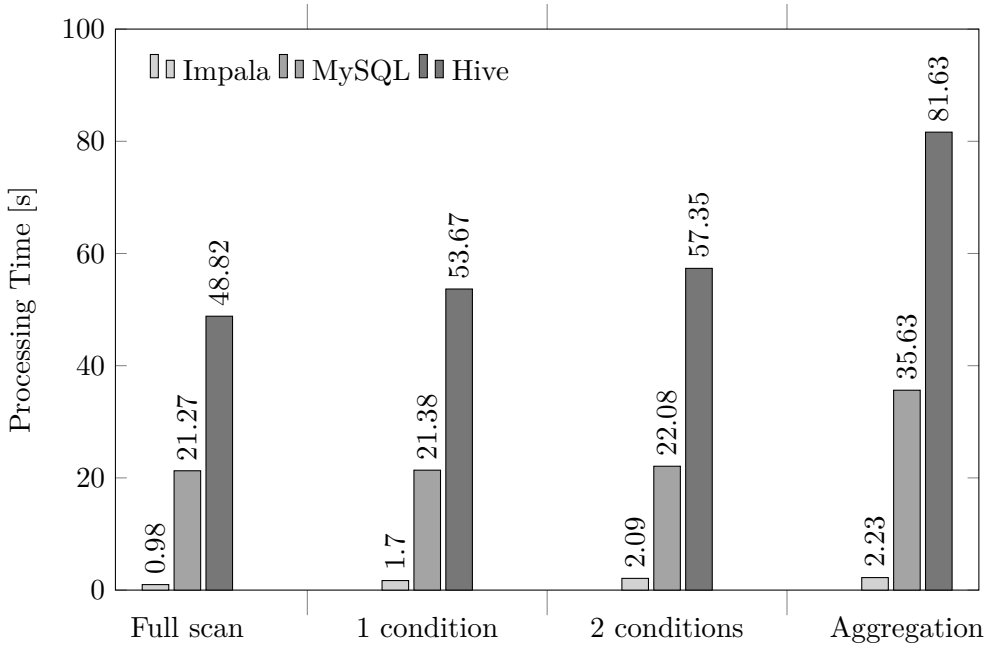


**Figure 5.6:** *Performance of Different Query Patterns*

As expected, Hive performs poorly compared to the other two, mainly because of the overhead *MapReduce* imposes on it. When datasets need to be aggregated, the overhead is even larger, because Hive needs to start multiple jobs to answer the queries. However, looking at the numbers of Impala, we are confident that we succeeded to choose the right architecture for our Framework. Furthermore, partitioning the data means that to answer queries of the Smart Clients, the engine will never need to scan significantly more samples than in our test. It is also very comforting, that aggregations can be done just as quick on the data as simple filtered queries, which we cannot say about the MySQL engine.

# Chapter 6

# Case Study

In order to observe the Framework from the Application developers' point of view, the idea of creating an application using the Framework, as a client application developer, struck us in the early stages of design. This was the best way to test the Framework capabilities, to provide a showcase application for application developers and to come up with new requirements towards the Framework. The following chapter covers the description of *BoardComputer* client application, capable of acquiring all available data using the Framework, and displaying it in an aesthetic form.

## 6.1 Initialization

To start using the Framework, the Data contract must be set. In this case the data contract contains all available data, and the *appID* is the name of the application. The data contract is defined in the following way:

```
{
        "appId" : "board_computer",
        "appName" : "Board Computer",
        "contract" : {
                "items" : [{
                                "measurementName": "Air Intake Temperature",
                                "measurementAlias": "air_intake_temp",
                                "measurementType": "REAL"
                        },
                        {
                                "measurementName": "GPS Time",
                                "measurementAlias": "gps_time",
                                "measurementType": "LONG"
                        },
                        ...
                        ]
        }
}
```

After this, the development of the mobile client application started. Before deploying any client application to a device, the Platform Application must be installed on it.

In order to use the client side part of the Framework, the Platform library was included in the following way in the build properties of the application, after copied into the library folder:

```
repositories {
        flatDir {
                dirs 'libs'
        }
}
dependencies {
        compile(name: 'vehicleict-platform-lib', ext: 'aar')
}
```

The application library was created and built with Gradle [35] build system, so in order to use the Platform this build system is required.

## 6.2 Connecting to the Platform

In order to connect to the Platform, we created a proper Configuration. The required parameters, except of the application name, are acquired by the client application itself. For *userID*, a login screen is created, and for *deviceName*, a settings screen lists the available devices. As the aim is to display all the available data, all the fields are set to be required (with the help of Fields class static method), and report interval is set 5 seconds, to give enough time for the Platform to read the data, but also display it frequently. As we planned to send measurement data to the server, reporting to the server is set to *true*.

```
Configuration configuration =
        new Configuration.Builder(BoardComputerConstants.APPLICATION_NAME, userID,
            deviceName)
                .setRequiredFields(Fields.getAllCommands())
                .setReportInterval(ReportInterval.Sec5)
                .setReportToServer(true)
                .build();
```

After the configuration, the Platform is created, and a callback interface is declared as an anonymous class. Then, the connection starts after the initialization.

```
Platform platform = new Platform(getApplicationContext(), configuration,
        new SampleListener() {
                @Override
                public void receiverSample(Sample sample) {
                        ...
                }
        });
platform.connect();
```

The initialization process happens when the measurement screen is created, and when this screen is destroyed, the platform disconnects. As connection process needs time on the Platform side, *startSampling()* cannot be called right away in the Client application. In order to avoid this, a start/stop measurement button is placed on the action bar of the application. This button

is only visible when the screen (Activity [13]) is created, and the creation of the screen needs more time than connecting, so the user is not able to start the sampling before the application is connected to the Platform.

In the event handler of this button, the sampling is started/stopped in the following way:

```
if (!isODBRunning) {
    platform.startSampling();
    isODBRunning = true;
} else {
        platform.stopSampling();
        isODBRunning = false;
}
```

## 6.3 Displaying the Gathered Data

When the sampling is running, the display process of the data fields is started in the defined callback method, called by the Platform with the actual Sample object. The data fields are displayed by using the setter functions defined by the application. For every field, its function is invoked the following way:

```
public void receiverSample(Sample sample) {
    boardComputerFragment.setEngineRPM(sample.getEngineRPM());
    engineDetailsFragment.setAirIntakeTemperature(sample.getAirIntakeTemp());
}
```

In all setter functions, the existence of the values are checked like in the server side forwarding (minimum value of type mean not existing data), and displayed if exists (or Not supported text appears). This filtering is done in the proper setter functions, in the following way:

```
if (position == Double.MIN_VALUE || position == Integer.MIN_VALUE ||
        position == Long.MIN_VALUE) {
        return false;
}
if (isAnimated) {
        this.startAnimating(position);
        this.position = position;
} else {
        this.position = position;
        this.invalidate();
}
```

On the next page, four applications are shown, which we have already created.


**Figure 6.1:** *Board Computer, Engine Details, Social Driving and Eco Driving*

# Chapter 7

# Related Works

As *connected car* services have a huge potential, it is no wonder, that there are other solutions all over the globe, that show similarities with our project. In this chapter, we investigate other *connected car* solutions both in the industry and in academic research, and highlight the attributes in our Framework that differentiate it from the others.

## 7.1 Industry Efforts

**CarCare** [43], a Hungarian solution, is a cooperation between Ericsson Hungary and Hungarian Telekom. This project also uses *On-Board Diagnostics* as the vehicle-side data source, with a special connected hardware. The device is capable of not just reading OBD data, but to acquire GPS positions as well. This special piece of hardware has a mobile internet connection capability, and uses it to send data to the backing infrastructure. The client device has only the role of data source, so *no immediate client features* are available.

**Connected Vehicle** [17] was made by two leading Swedish companies: Volvo car company in cooperation with Ericsson. This is a unique solution, as it uses a custom Client device, built into the cars as an on-board user interface. This device has its own operating system (based on a previous Android version) that hides all the details of the vehicle- and server-side communication. This solution also uses the Cloud as the base of its backing infrastructure, and currently supports applications like *Road usage charges*, *Traffic safety* and *Infotainment*. Although the Cloud service offers user features, their user interface is only the custom board device, so the usage of these services is locally bounded to the car. This solution only supports cars manufactured by Volvo. However, they are planning to expand the ecosystem, in cooperation with other vehicle companies.

**Delphi Connect** [12] is an already existing *vehicle-management* service, provided by Verizon, one of the largest telecommunication companies in the United States. This solution uses a unique *LTE-capable* [39] device, connected to the car through OBD II. This device has similar OBD II related functions, like the previously described hardware, but it has interesting extended features,

too. Unlike the previous solutions, this one is capable of not just *vehicle-to-device/infrastructure* communication but in the other direction, too. Although users are not capable of driving the car remotely, but security features like locking/unlocking the car and using the horn are supported, users are even able to start the engine. As this solution only provides *vehicle-management* features, it is not intended to be extendible, and cannot be used as a Framework or a base of other services.

Besides the previously mentioned solutions, there are several projects working on *connected car* services. However, most of the technical details of these solutions are not public. *Snapshot* [48] a smart car insurance service by Progressive (USA) is using OBD II to calculate fees. Dialexa Labs offers *Vinli* [41], a *vehicle-management* service supporting not just smartphones, but *smartglasses* (like Google Glass), too. *SmartDrive* [50], a Hungarian startup has developed its own hardware to access internal vehicle data, and it is currently building a *social game* based on *connected cars*.

## 7.2 Academic Research

There is also strong academic research in the connected car domain, for example [10] describes the Cloud-related aspects of *connected car* applications, and [38] also depicts a Cloud-based solution using OBD II.

The solution described in [10] focuses directly on the infrastructure, and all vehicle-related aspects are the responsibility of application developers. Instead, this solution dives into a design of a *PaaS* (Platform-as-a-service) system, and investigates wide variety of features for *connected car* applications, such as *collision avoidance*, *theft control*, *lane change* support and *pedestrian safety*.

Paper [38] also focuses on the client side, and unlike previous solutions, it uses an Android mobile device as a client-side unit, like our Framework, and uses OBD II for sampling. This project applies Cloud computing as the backing infrastructure, but no additional service support details are public.

## 7.3 Assessment

We have not yet found a solution on the market or in the academy, that focuses strongly both on consumer-side applications and background services. It is also hard to find efforts, which create universal frameworks, that are capable of hosting a diverse set of services, and are easily extendible. We think, that this is the real value of our Framework, compared to other solutions. However, a limit of our Framework in its current state is, that it depends on smartphones and tablets the driver owns, and not capable of communicating with the vehicle without those. As we placed a strong emphasis on Client applications, we consider this an opportunity, and not a deficiency. We are also proud, that unlike many other solutions, ours is capable of collecting data both from OBD II and the CAN bus of the vehicle.

# Conclusion

In our paper, we have described the *VehicleICT Framework* that has been created to allow a number of teams to develop connected car services with infrastructure-side support for data processing.

We have developed a Platform, on which Android applications can be built that are able to use the Infrastructure. We have worked out a solution, which allows multiple applications to run on the same device and benefit from the features of the Platform. We have also tested the Platform by implementing actual applications, which can also be used as examples to service developers.

To make the large amount of data collected from the vehicles available for querying and analyses, we have introduced a standard dataflow that is applied to the data of all services individually. To implement this dataflow efficiently, we have built a Hadoop cluster, which holds the datasets and enables their high-performance processing. We have included extension points in the dataflow, to make it possible for service developers to customize it with their own transformation logic or create their own data structures specific to their service. We have also created a Mediator Agent, which sits in the center of the architecture and aids the communication between the Smart Clients and the Data Center. We have shown, that the Framework is able to provide high-performance, low-latency querying of the datasets, which is crucial to client applications and is also beneficial to reporting applications. We have designed the Infrastructure with attention to the possibility that later we might need to scale up, so the Framework is portable to the Cloud.

Our Framework is already used by four different teams, who are building connected car services. We take this as a confirmation, that our approach has a reason for existence, and our Framework is capable of providing the support that service developers need, which was our intention.

This work summarizes our experiences that led us to the working Framework, that was released in parallel of this research. Our proposed solution has two major components: a server-side component applying Big Data techniques, and a universal platform solution for mobile devices. These two modules are the base of the Framework. This research report can be considered as a pioneering work in the field of universal frameworks for smart car solutions.

We believe, that *VehicleICT Framework* makes it possible to create services, which transform the way people drive their cars. We are also confident, that our research, the concept we have come up with and the experience we have gained in the process are not restricted to the connected car domain, and we shall be able to extend the Framework in the future to accommodate services in other domains, too, such as the emerging *Internet of Things*.

# List of Figures

# List of Tables

# Bibliography


[1] Amazon. Amazon Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce/`, October 2014.

[2] Amazon. Amazon web services. `http://aws.amazon.com/`, October 2014.

[3] Ralph Birnbaum and Jerry Truglia. *Getting to Know OBD II.* A S T Training, 2000.

[4] Bluetooth SIG. What is Bluetooth technology? `http://www.bluetooth.com/Pages/what-is-bluetooth-technology.aspx`.

[5] CAN in Automation. Controller Area Network (CAN). `http://www.can-cia.de/index.php?id=systemdesign-can`.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*. Seattle, WA, 2006.

[7] Cloudera. Cloudera Impala. `http://impala.io/`, October 2014.

[8] Cloudera, Databricks, IBM, Intel, and MapR. Community effort driving standardization of Apache Spark through expanded role in Hadoop projects. `http://www.cloudera.com/content/cloudera/en/about/press-center/press-releases/2014/07/01/community-effort-driving-standardization-of-apache-spark-through.html`, October 2014.

[9] Alistair Cockburn. Hexagonal architecture. `http://alistair.cockburn.us/Hexagonal+architecture`, October 2014.

[10] Sohrab Modi David Bernstein, Nino Vidovic. A cloud paas for high scale, function, and velocity mobile applications - with reference application as the fully connected car. In *Systems and Networks Communications (ICSNC), 2010 Fifth International Conference on*, pages 117 – 123. Conference Publishing Services, Nice, France, 2010.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004.

[12] Delphi. DelphiConnect. http://delphiconnect.com/.

[13] Android Developers. Activities. http://developer.android.com/guide/components/activities.html.

[14] Android Developers. Android Interface Definition Language (AIDL). http://developer.android.com/guide/components/aidl.html.

[15] Android Developers. Bound Services. http://developer.android.com/guide/components/bound-services.html.

[16] Spring Framework Documentation. Spring MVC framework. http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html, October 2014.

[17] Ericsson. Connected Vehicle - an industry in transformation. http://www.ericsson.com/thecompany/press/mediakits/connected-vehicle.

[18] The Apache Software Foundation. Apache Flume. http://flume.apache.org/, October 2014.

[19] The Apache Software Foundation. Apache Hadoop. http://hadoop.apache.org/, September 2014.

[20] The Apache Software Foundation. Apache HBase. http://hbase.apache.org/, October 2014.

[21] The Apache Software Foundation. Apache Hive. https://hive.apache.org/, October 2014.

[22] The Apache Software Foundation. Apache Hue. http://gethue.com/, October 2014.

[23] The Apache Software Foundation. Apache Mesos. http://mesos.apache.org/, October 2014.

[24] The Apache Software Foundation. Apache Oozie. http://oozie.apache.org/, October 2014.

[25] The Apache Software Foundation. Apache Parquet. http://parquet.incubator.apache.org/, October 2014.

[26] The Apache Software Foundation. Apache Pig. http://pig.apache.org/, October 2014.

[27] The Apache Software Foundation. Apache Spark. https://spark.apache.org/, October 2014.

[28] The Apache Software Foundation. Apache Sqoop. http://sqoop.apache.org/, October 2014.

[29] The Apache Software Foundation. Spark Streaming. https://spark.apache.org/streaming/, October 2014.

[30] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests.* Addison-Wesley, 2010.

[31] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *19th ACM Symposium on Operating Systems Principles.* ACM, Lake George, NY, 2003.

[32] Google. google.gson. `https://code.google.com/p/google-gson/`.

[33] Google. Google cloud platform. `https://cloud.google.com/`, October 2014.

[34] Google. Memcache java api overview. `https://cloud.google.com/appengine/docs/java/memcache/`, October 2014.

[35] Gradle.org. Gradle. `http://www.gradle.org/`.

[36] Hortonworks. Hadoop Distributed File System. `http://hortonworks.com/hadoop/hdfs/`, September 2014.

[37] ECMA International. The JSON Data Interchange Format. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

[38] Shi-Huang Chen Jheng-Syu Jhou. The implementation of obd-ii vehicle diagnosis system integrated with cloud computation technology. In *Intelligent Data analysis and its Applications, Volume I*, pages 413–420. Springer International Publishing, Shenzhen, China, 2014.

[39] LTE World. LTE Advanced: Evolution of LTE. `http://lteworld.org/blog/lte-advanced-evolution-lte`.

[40] Mark Stammers. Eobd Codes explained. `http://www.pikit.co.uk/peugeotmt/eobd%20p%20codes%20explained.html`.

[41] Matt Burns. Vinli Promises To Bring Autos Into The Smartphone Age. `http://techcrunch.com/2014/09/08/vinli-promises-to-bring-autos-into-the-smartphone-age/`.

[42] Microsoft. Open database connectivity overview. `http://support.microsoft.com/kb/110093`, October 2014.

[43] Molnár József. Car Care - facebookozik a kocsid. `http://pcworld.hu/eletmod/car-care-facebookozik-a-kocsid.html`.

[44] Hungarian Central Statistical Office. Motor-cars in Hungary. `http://www.ksh.hu/docs/hun/xstadat/xstadat_hosszu/h_odme001.html`, September 2014.

[45] Oracle. Trail: The Reflection API. `http://docs.oracle.com/javase/tutorial/reflect/`.

[46] Oracle. The java persistence api. `http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html`, October 2014.

[47] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178. ACM, New York, 2009.

[48] Progressive. Snapshot. `http://www.progressive.com/auto/snapshot/`.

[49] Roy Thomas Fielding. Representational State Transfer (REST). `http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`.

[50] Tóth Balázs. Utcai autóverseny a biztonságos vezetésért. `http://index.hu/tech/2014/10/13/utcai_verseny_a_biztonsagos_vezetesert/`.

[51] W3C XML Working Group. Extensible Markup Language (XML) 1.0. `http://www.xml.com/axml/testaxml.htm`.

[52] Wikipedia. Cloud computing. `http://en.wikipedia.org/wiki/Cloud_computing`, October 2014.

[53] Wikipedia. Push technology. `http://en.wikipedia.org/wiki/Push_technology`, October 2014.