A Novel IoT Platform for the Era of Connected Cars

Attila Csaba Marosi and Róbert Lovas
Laboratory of Parallel and Distributed Systems,
Institute for Computer Science and Control,
Hungarian Academy of Sciences
H-1518 Budapest, P.O.Box 63, Hungary
Email: {attila.marosi, robert.lovas}@sztaki.mta.hu

Ádám Kisari and Ernö Simonyi
Systems and Control Laboratory,
Institute for Computer Science and Control,
Hungarian Academy of Sciences
H-1518 Budapest, P.O.Box 63, Hungary
Email: {adam.kisari, erno.simonyi}@sztaki.mta.hu

***Abstract*—The race among manufacturers to build convenient, safe, and autonomous Connected Cars by applying the latest digital technologies, and ultimately a completely self-driving vehicle, is already underway. One of the cornerstones of such vehicles is the continuous ingestion of massive amount of data from wide variety of hardware components, including sensors, on-board cameras, and further external sources. Cloud computing and big data processing are ideal candidates and already proven technologies in order to store and process the heterogeneous, rapidly growing, and large-scale data sets. The cloud may act as a kind of central hub or as an Internet of Things (IoT) back-end where the sensor and the other available data can be gathered while also offering an elastic platform where the vast amount of data can be processed, analyzed and distributed real-time. In our paper we detail the evolution of a cloud-based, scalable IoT back-end framework and services built on top for handling and processing vehicular data in various use case scenarios: CAN data collection, remote device flashing, Eco-driving, weather report and forecast. The first version is an Infrastructure-as-a-Service (IaaS) solution with a reference implementation deployed on an OpenNebula based cloud. The second iteration runs on a private Platform-as-a-Service (PaaS) cloud built on the Cloud Foundry platform within the premises of an automotive supplier company. Both variants have been successfully evaluated and validated with benchmarks.**

I. Introduction

Connected car technologies have been racing ahead as vehicle manufacturers continue to unveil newer digital services and autonomous driving features. Connected cars constantly collect and make sense of massive amounts of data from a huge array of sources. They talk to other cars, exchange data and alert drivers to potential collisions. They can also communicate with sensors on signs on stoplights, bus stops, and even ones embedded in the roads to get traffic updates and rerouting alerts. And lastly, they can communicate with your house, office, and smart devices, acting as a digital assistant, gathering information you need to go about your day. Even though automobiles today contain an impressive amount of processing power, the amount of information flowing back and forth inside them requires technology with considerable storage capabilities that can handle sophisticated processing and analytical functions [1] [2]. An ideal task for cloud computing [3] and big data [4] processing, used by cars on the road. In our paper we detail the evolution of a cloud based framework and services built on top for handling and processing vehicular data. The first version is an Infrastructure-as-a-Service (IaaS) based solution with a reference implementation deployed on an Open Nebula based cloud. The second iteration runs on a private Platform-as-a-Service (PaaS) IoT cloud built on the Cloud Foundry platform within the premises of an automotive company.

The paper is structured as follows. Section II discusses related work. Section III discusses the different architecture variants and their reference implementations. Section IV details the different functionalities and applications built on top of the architecture. Several smartphone applications were developed for their respective functionalities, these are also described in this part of the paper. Section V evaluates our framework, while section VI details future work and concludes the paper.

II. Related work

Amazon Web Services (AWS), Microsoft Azure and Google Cloud may be considered the three dominant forces in public cloud computing, and all three provide their own IoT platform and services [5], [6], [7]. These are generic, hosted platforms and not available as an on premise private solution. There are several proposals available for big data processing that aim to provide a generic architecture rather than one that fits a single use-case [8], [9], [10]. Next we are going to discuss two such architectures:

The lambda architecture for Big Data was proposed by Nathan Marz and James Warren [9]. It aims to decompose computing arbitrary functions on arbitrary dataset in real time into three layers: (i) speed layer; (ii) serving layer; and (iii) batch layer. Its batch layer is responsible for precomputing results applying a distributed processing platform that is able to handle extremely large quantities of data. The main goal of this layer to provide high level accuracy with its processing capabilities on all available data when generating views, i.e. this layer is able to fix any errors using the complete data set (and later updating the already existing views). Outputs are mostly stored in a read-only database, with updates completely replacing existing precomputed views. Hadoop [11] is the de facto standard for batch-processing in most high-throughput

systems. The speed layer can process data streams in real-time even without fix-ups or completeness of such streams. As a trade-off, the speed layer sacrifices throughput when it provides real-time views on the latest data in order to minimize latency. In other words, the speed layer is responsible for somehow filling the "gaps" originated from the previous layer's lag. The generated views may not be accurate or complete, on the other hand, such views become available almost immediately after receiving the data (might be replaced when the batch layer's more accurate or complete views for the same data become ready). Stream-processing technologies typically used in this layer include Apache Storm [12] or possibly Apache Spark [13], while output is typically stored in NoSQL databases like Cassandra [14]. Outputs from both layers are stored in the so-called serving layer that is responsible for responding ad-hoc queries either by providing precomputed views or building views. Dedicated stores are used in the serving layer, e.g. Apache Cassandra or Apache HBase [15] for speed-layer output, and Cloudera Impala[16] for batch-layer output. The architecture emphasizes the problem of reprocessing data thus, processing input data over and over again. This is required since applications using the data may contain bugs that produce incorrect results, and after fixing them the data must be reevaluated; or the application simply evolves and new outputs are required from the same data. The lambda architecture (or a variant of it) is in production by Yahoo for analytics on its advertising data warehouse and by Metamarkets [17].

The FIWARE Big Data Architecture [8] was created within the FIWARE (Core Platform of the Future Internet) EU funded R&D project as one of many Generic Enablers (GEs). A GE is "*a functional building block of FIWARE. Any implementation of a FIWARE GE is made up of a set of components which together supports a concrete set of Functions and provides a concrete set of APIs and interoperable interfaces that are in compliance with open specifications published for that GE*" [18]. The Big Data GE architecture expands the basic Apache Hadoop one. The Master Node has all management software and acts as a frontend for the users. Infinity is the permanent storage cluster (based on HDFS). Computing clusters have a lifecycle: they are created, used for computation and finally they are removed. All data must be uploaded to Infinity beforehand. Data can be uploaded to and retrieved from Infinity via WebHDFS [20] or Cosmos CLI (a command line interface to WebHDFS). The Big Data GE specifies the use of SQL-like analytics tools like Hive, Impala or Shark. Although the GE is based on Hadoop, FIWARE proposes several alternative options: (i) Cassandra File System can be used instead of HDFS and still Hadoop can be used on top of it; (ii) a distributed NoSQL database like HBase can be installed on top of HDFS; (iii) use Cascading [21] as an extension or replacement. FIWARE Big Data GE was created with multiple parallel users in mind. For each user or calculation a dedicated environment (cluster) is deployed and later tore down.

There are also specific efforts for creating cloud-based solutions for Connected Cars, one of them (with wider and slightly different scope than ours) is the Connected Car Prototyping Platform [1]. This pattern based platform builds upon a layered architecture similarly to our approach (see Section III) and their implementation using also a widespread platform, namely VMware. However, they omit several crucial features, such as high availability at application level and performance benchmarking, since they did not consider them necessary for prototyping.

Another Connected Car specific effort describes Vehicular Data Cloud Services in the context of an IoT environment that combines not only IoT platforms but traditional cloud services with so-called temporary clouds as well [2]. In their community-based approach, such temporary clouds might be formed by harvesting the underutilized (spare) IT capacities of vehicles, i.e. storage space, network bandwidth and computation power, in order to provide intelligent parking facilities for drivers. However, they emphasized several open issues as well (e.g. security) which may make the usage of their system in real circumstances impossible. Our goals remain at traditional IaaS and SaaS level with investigating other uses cases (see Section IV).

III. Architecture

The goal of the framework is to reliably receive and store incoming sensor data from multiple array of configured sensors with the capability to scale as the number of sensors (and the incoming data) grows. This is augmented by different user facing and administrative applications.

Sensor data is usually generated by small microcontroller based devices where usually raw data is from one or many different instruments. Measurements are taken periodically and thus it generates a lot of small packets that usually consist of instrument, sensor id, node id, timestamp, and measured data. Storing large volume of this kind of data requires a tailored infrastructure with the capability to scale up (horizontally) as the volume of data grows. The architecture follows a three-tier layout as depicted in Fig 1. Each component of each tier is typically deployed on a separate node. This allows easy scaling of appropriate tiers for example when the volume of incoming sensor data increases or decreases.

Fig. 2 depicts the generic architecture of applications implementing functions. The figure contains two archetypes for Applications. *a* in Fig. 2 details the Web Application Architecture (WAA) and *b* in Fig. 2 details the TCP Application Architecture (TAA). Both architectures are built using three major blocks:

1) Security: Secure channel between clients and the application; authentication and authorization; database access and function specific requirements.
2) Application: The function implementation and required components (Framework, Object Mapper, etc.) that implements the given function logic.
3) Database: Interaction with the internal (to the cloud) Database Service (including the Object Storage Service).

Next WAA and TAA generic characteristics are discussed:

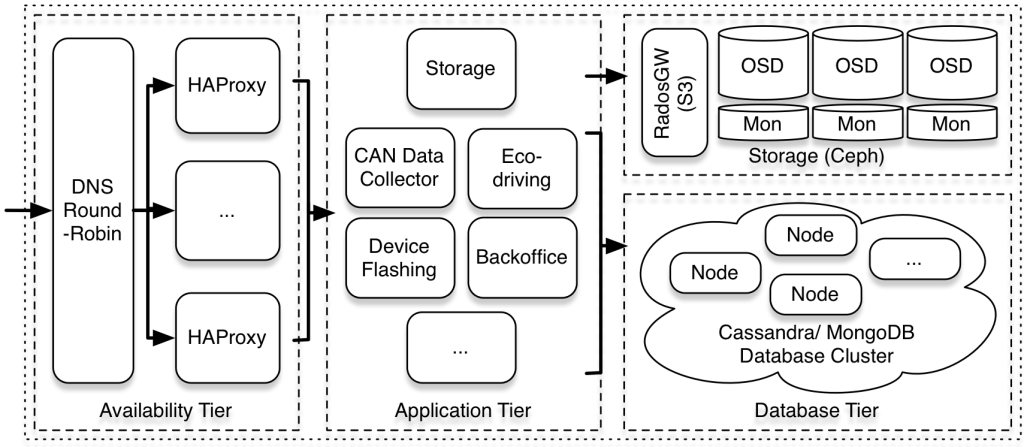


Fig. 1: Three tiers of the architecture

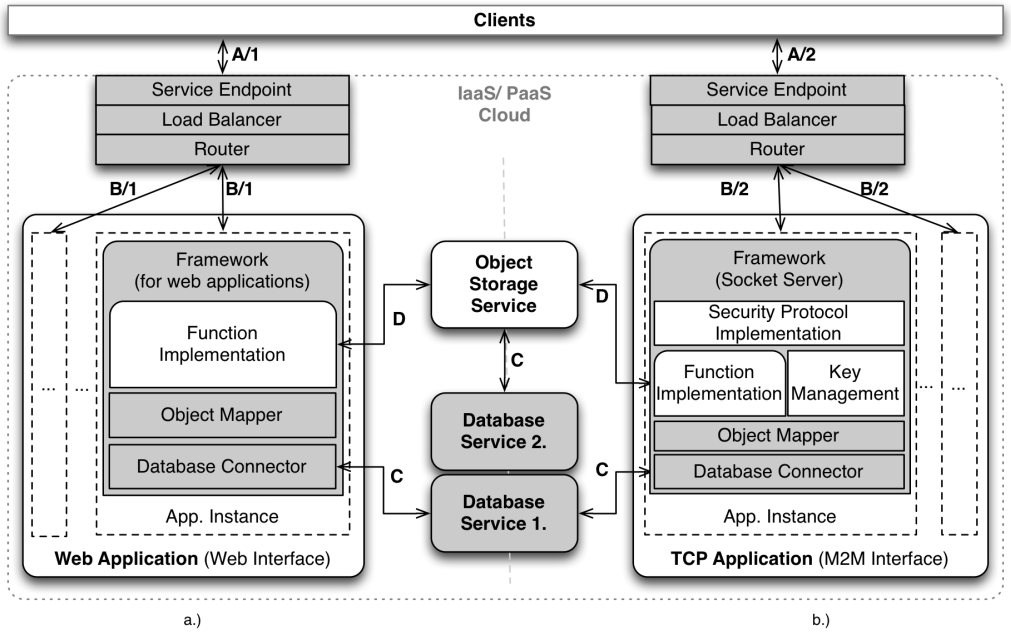


Fig. 2: Generic architecture for a.) Web; and b.) TCP applications; with their connected components

1) Web Application Architecture: WAA (presented in Fig. 2) is for functions that expose a cloud based user interface for the users (e.g., Backoffice or Community Based Weather Forecast). Security (see *1* in Fig. 2) refers to HTTPS traffic between the Service Endpoint and connected clients (i.e., web browsers or mobile applications). This HTTPS traffic is then directed to an application instance by the load balancer and routed to the internal service endpoint of the instance (represented by *B/1* in Fig. 2). This internal (to the cloud) traffic is still HTTP however; it is not encrypted anymore (SSL/TLS is terminated at the frontend). Authentication is user and password based (over the secure channel). Usually the web framework provides a means for storing the user password securely (i.e. salted and encrypted). In case of Flask this is provided by Flask-Security module. Authorization in WAA based application is role based. An Application (implementing a function) may have multiple instances. This is required for load balancing and failure tolerance. These instances are identical and can be discarded any time thus; they cannot contain any user-uploaded data. An application contains the implementation of a single function, however it is possible for a function to have a user interface (WAA) and machine interface (TAA). This is detailed at the functions. A WAA based application builds on a web application framework (e.g. Flask). The framework also provides the Object Mapper and Database Connector that allow interacting with the database service. The Database component is responsible for containing all persistent data, as the application instances cannot store them locally. The connection with the database service (see *C* in Fig. 2) is internal (to the cloud) but depending on the

selected database type can be secured as well. Each application has unique credentials for the database service so it can access only its data. The exact requirements will be determined during implementation. The Object Storage Service (OSS in Fig. 2) is a special *database service* as it is intended for storing binary data (*blobs* e.g., for the Device Flashing function). Whenever binary data transfer is detailed at any function it is assumed that the meta data is stored in the database and the blob is stored in the object store. It is directly connected with application instances (see *D* in Fig. 2) however it is intended only as an overlay above a database service. Credentials for accessing it are also unique to each function and enforced by the database service. WAA based applications are considered by default compute intensive as they provide a dynamic rich web based user interface with additional functionality (e.g., reporting) that requires even more compute capacity. For the prototype introducing additional application instances can solve the problem. For future work decoupling frontend (e.g., displaying the user interface) and backend (e.g., generating reports) functionalities is required. Additionally WAA based applications can be network intensive as well. If this is the case we add a note at the given functionality.

2) TCP Application Architecture: TAA is presented as Fig. 2/b. It is similar to WAA with the major difference stemming from the difference in communication channel (TCP vs. HTTPS). Channel encryption, authentication and authorization are provided by a developed protocol that is out of scope for this paper (depicted as *Security Protocol Implementation* in Fig. 2/b). *A/2* in Fig. 2 depicts the encrypted channel; in this case the encryption should extend to the application instance (i.e., *B/2* is encrypted as well) as the protocol is implemented at the application level. As authorization is certificate based (in case of Public Key Infrastructure [PKI]) the Key Management component is responsible for storing keys. Whether it is a local store (wired in to the application instance) or uses a database backend is not in the scope of the architecture document. For TAA based applications the Application itself cannot be based on a web framework, as it does not expose a web interface. Rather it exposes a native TCP interface that can be implemented at low level or with a support of a framework (e.g., Socket Server in Python). Database and OSS access is not different from WAA. TAA based applications are considered network intensive since they produce no user interface only data exchange is involved between peers. However this insensitivity can be continuous or burst. By default we assume continuous and we note for each function discussed if it has burst like characteristics. TAA based applications can be compute intensive if they include some mean of processing. This is also noted in the summary table of the function. Please note that for future work the frontend and backend functionalities should be separate.

A. Variant 1: Infrastructure level architecture

The High Availability and Load Balancing Tier (shown in Fig. 1) accepts incoming sensor data and forwards it to one of the data collector application instances in the Application Tier. The forwarding decision is made in two steps. First based on a round-robin algorithm a high-availability proxy and load-balancer (currently based on HAProxy [23]) is selected. The proxy in turn will select an application server with the lowest load and forward the request to that one. A CAN Data Collector instance in the Application Tier (shown in Fig. 1) will decode the received data and store them in the Database Tier (shown in Fig. 1). Besides the CAN Data Collector, other functionalities (see section IV for more details) are also available and work similarly.

The Database Tier consists typically of a Cassandra or MongoDB database cluster, besides a RDBMS like MySQL. Cassandra is a decentralized structured storage system that is well suited for storing time-series data like connected car sensor data. As the volume of incoming data changes Cassandra allows dynamically adding or removing new nodes to the database cluster to either scale up or down.

Meta data submission is initiated by resolving the DNS endpoint. The DNS endpoint may contain one or more load-balancer addresses, in turn they distribute the load between the available Receiver instances. Using round-robin DNS techniques, it is possible to scale the number of load-balancer nodes. Round-robin DNS is a well-known simple method for load sharing, fault tolerance and load distribution for making multiple redundant service hosts available. In the simplest implementation round-robin DNS returns a list of IP addresses. For each such request the returned list is permuted and the client will connect to the first address in the list. If the connection fails the second address should be tried and so on. The permuting provides even distribution of requests between the addresses in the list. If a server from the returned list fails, the requests towards it will timeout and the client should try the next server on the list. This will result in longer service handling times, but ensures that the requests are served by one of the working servers. Sequential requests can and usually will be served by different servers thus, if the application in the Application Tier is not stateless (e.g., has user sessions) then either all requests belonging to the same session must be handled by the same server (not possible when Round-Robin DNS is used) or the Application Tier must be prepared to share session data between all application servers. The receiver is stateless so no such measures are required and round-robin DNS is well suited.

HAProxy servers are responsible for balancing the load across multiple application servers (e.g., CAN Data Receivers) after through the Round-Robin DNS the client contacts one. HAProxy is a lightweight high-performance TCP/HTTP load balancer. It continuously monitors the health and performance of the application servers connected to it and proxies the submission request to the least loaded one.

The Application Server Tier is depicted in Fig. 3 and presented via the CAN Data Collector application. It consists of the following: Chef is used as a deployment orchestrator for bootstrapping new nodes for the different tiers. The Data Processing component and Cassandra Connector are implemented using the Flask Web Framework and Python. The

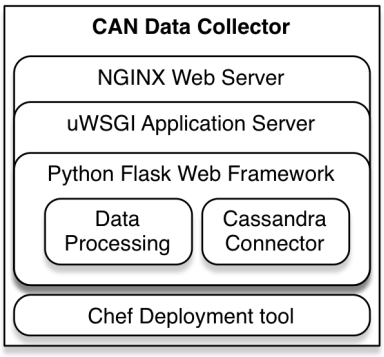


Fig. 3: Architecture of the CAN Data Collector

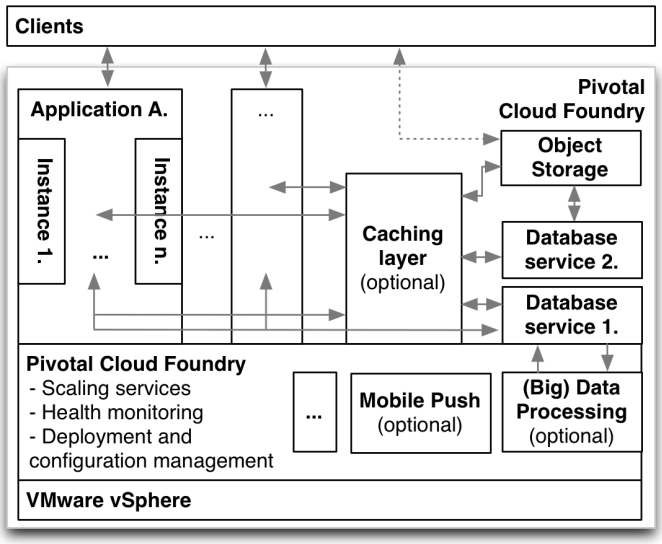


Fig. 4: Architecture on Cloud Foundry

Sensor Meta Data Decoder is responsible for interpreting the incoming data and passing it to the Cassandra Connector. The Cassandra Connector is used to store the decoded meta data in the database cluster. uWSGI [25] is used as a WSGI [24] application server, and finally NGINX [26] is connected to the wire-protocol of uWSGI to achieve a high performance WSGI based web frontend. Other applications have (e.g., Backoffice) similar architecture.

B. Variant 2: Platform level architecture

Cloud Foundry [27] allows to create on premise Platfrom-as-a-Service (PaaS) clouds. It is available as an open source software stack and from many commercial vendors either as an on premise (e.g., Pivotal) or hosted solution (e.g., part of IBM BlueMix). We adapted our platform for an on premise, IoT cloud based on a private deployment of Pivotal Cloud Foundry (PCF) running in a data center of an international automotive company. Fig. 4 depicts the resulting architecture at a high level with the components as follows.

Object storage (see Fig. 4) provides a mean to store and retrieve (or publish) large binary data blobs. It can be used for e.g., distribute publicly available files. This service can be infrastructure level, e.g., providing storage for the virtual machines, or platform level, e.g., providing a REST interface for storing and downloading files. For Variant 1 (see section III-A this functionality is provided by Ceph [28] through its Amazon S3 compatible RadosGW service. PCF does not recommend storing data in application instances, since that is local only to the instance and is not permanent and instances do not share a file system. PCF did not provide such a functionality, thus this means that either such a component needs to be developed or the use of an external service is required. To keep all components on premise and also since sensitive data may be stored in the Object Storage we rolled our on solution.

Configuration management and deployment service is provided by PCF. Environments and different services (e.g., application scaler or database services via marketplace) can be provisioned within the platform. At the infrastructure level (see Variant 1 in section III-A) Chef [29] is utilized.

High-availability, load-balancing and health checking services refer to the capability of the system to tolerate service failures and balance traffic between available services and with configuration management and deployment provide horizontal scaling capabilities for the system. PCF allows applications to scale horizontally and vertically. For marketplace services (e.g., database servers) it highly depends on the service (e.g., allows horizontal scaling) and the available service plans.

There are two database services: The first one (see Database service 1. in Fig. 4) is for storing application data; and the second database service is for the object storage service (see Database service 2. in Fig. 4). The first is MongoDB (Cassandra was not supported by the IoT cloud), while the second is MySQL for storing meta data for the object storage.

The role of the caching layer is storing frequently accessed data. From the PCF service catalogue typical services used for implementing the caching layer can be Redis or Memcached.

Each application implements a functionality (see section IV). Scaling is either by increasing the number of instances for the application (horizontal scaling) or increasing the resources available for given instance(s) (vertical scaling). PCF uses an own implementation for traffic routing. Although it is possible to deploy a custom load-balancer like HAProxy by the administrators, however it is not recommended for production purposes. The dashed line between the Clients and the different Applications (in Fig. 4) represent the communication channel and protocols. The channel is HTTP for higher-level APIs and end user services, while they can be e.g., TCP for low-level services (assuming PCF platform allows it). However these and the application layer protocols are specific to applications and thus, will not be detailed here. Authentication and authorization is application specific. For sensitive services (i.e., CAN data collector) a security protocol is needed. For end user services like the Connected Car Backoffice System (see section IV-A), password based authentication and role based access control can be used over HTTPS. This can be extended e.g., by two-factor authentication using e.g., an SMS gateway, but this is beyond the scope of this paper.

IV. Functionalities (Use cases scenarios)

In this section we detail the functionalities of our platform through use case scenarios.

A. CAN Data Collector

The core function responsible for the collection and storage of several CAN (Controller Area Network) messages in the Cloud for further processing. It perfroms a basic extract, transform, load (ETL) functionality by pre-processing the data and storing it in a structured format for the other cloud-based functions. Additionally it is also stored directly in the input format to have backup for security reasons and to be available for future implemented functions and statistics. The communication between the smartphone application and the cloud service uses a secure connection as the data may contain private information (e.g. location). Within the encrypted channel the plain data is sent in a JSON format. There are two methods used in this function to communicate between the car and the cloud: periodic and event driven upload. These two methods can be combined: it is possible to set some parameters to be collected periodically while other parameters are sent in an event driven fashion. It is sensible to use periodic mode for telemetry type parameters and to use event driven mode for special features such as ABS or ESP activity or in the case of an accident (airbag or impact sensor). At the first connection or on special request the cloud service downloads the settings to the smartphone of the user. These setting are the list of selected parameters to collect from the vehicle, the periodicity of the data collection, and the connection type including the periodicity of the upload to the cloud. With the settings present on the device, if the connection is established with the vehicle the smartphone can send these settings to the vehicle. In case of the periodic data, Fig 5a illustrates the vehicle to cloud via smartphone communication. After the initial setup the data collection is triggered by the engine start. From this point the car sends all selected parameters to the smartphone with the given periodicity. The phone aggregates this data and uploads it to the cloud service with a longer periodicity. This arrangement optimises the communication, improving battery life and server load.

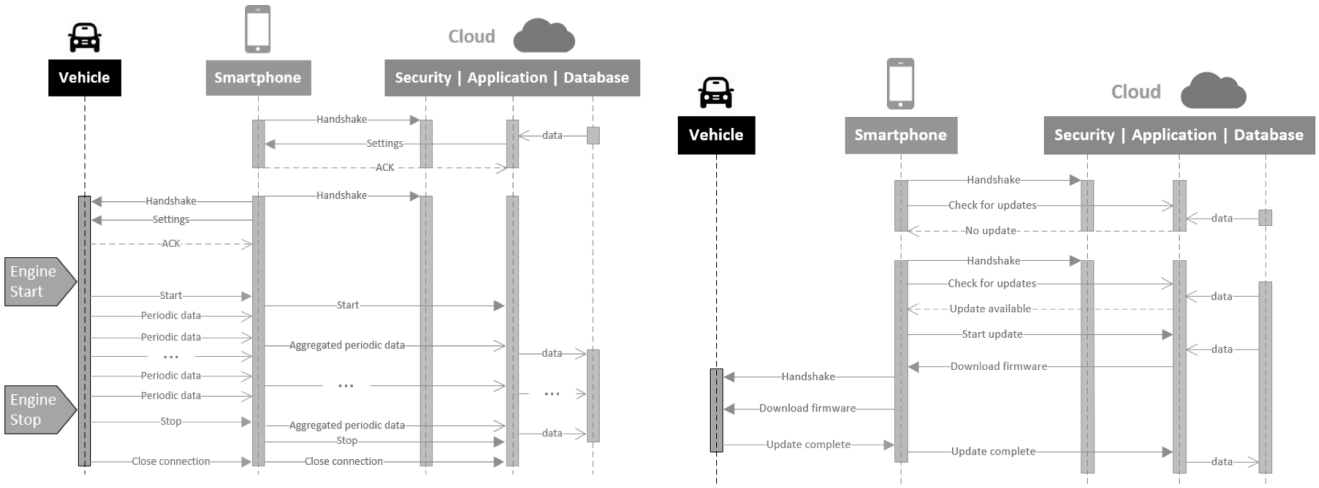
In case of the event driven scenario, after the initial setup, which is the same as in the previous method, each selected event is immediately sent to the cloud service through the smartphone. In this case the smart device doesnt do any kind of processing on the data, it works simply as a relay. If the vehicle is connected to the cloud without the smartphone, by an internal GSM module or any type of V2X communication system, periodic data collection is not available. The onboard computers dont have the necessary storage space to aggregate the data without the smartphone. This way the measured data is sent immediately, resulting in the same scenario as the event driven type.

B. Device Flashing

This function provides firmware upgrade capabilities for the Connected Car system. Updating a device over a cloud service raises several security concerns. Access to the firmware files should be limited to authorized personnel only. Different files may require different authorization thus specific roles may be required, e.g., a service partner may get access or even a car owner can be considered authorized in different scenarios. Usually an update is initiated by a person connected to the car. In this case the system has to make sure the car is safely parked and the battery will provide enough power for the whole process. Situations when the connection is lost during an update should also be managed. Some security updates however may be published as mandatory updates and should be downloaded to connected cars regardless of the owners initiation. In this case the car also has to be parked safely and in absence of the owner a direct GSM connection is needed. If the owner is using the car when the system wants to attempt an update, then the system should wait for an appropriate and safe time to start the process.

The cloud service needs to implement a way to distribute the updates amongst the cars. This is not trivial because even if an update package contains a small amount of data, it will be delivered to millions of cars, all over the world. The technology that should be used is a Content Delivery Network (CDN). Most of the publicly available cloud service providers offer such a service. This can be part of a rented infrastructure or it can be a hybrid solution where only the CDN service is outside the on premises, private cloud architecture. To deliver the flash data to 1000 cars in a region in 1 minute would require $\sim$160Mbit/s bandwidth, but in 1 hour only $\sim$26.49Mbit/s. For large scale deployments this scales linearly, i.e., for 100 000 cars within 1 hours would require $\sim$2649Mbit/s bandwidth. Serving from a single source, even within a region is not feasible for large scale deployments and/ or the time frame must be increased.

Additionally the delivered data should be encrypted to prevent unauthorized access and to protect data integrity. This way the on-board computer of the car can validate the origin of the retrieved data and make sure it was not modified by malicious attackers or network errors. After the data is delivered to the smart device it needs to be securely forwarded to the vehicle. The device flashing process can start after the data is fully downloaded, unencrypted and verified. In case the vehicle is connected directly to the cloud without the smartphone, by an internal GSM module or any type of V2X communication system, one layer of security can be removed from the data transfer. In this scenario the smartphone acts as a user interface. In Fig. 5b the scenario is illustrated when the phone communicates with both the cloud and the car, and initiates the update. Another scenario is also possible depending on the facilities of the vehicle: if there is no direct communication between the car and the smartphone, the cloud will initiate the update of the car after the phone sent the signal to the cloud. We assume the following products with corresponding data size for flashing with new software: (i) Body Computer (BCM) using 1 MB of data; (ii) Electric Battery Sensor (EBS) using 64 KB of data; and (iii) Parking assistant with 512 KB of data. Although difference in ECU



(a) Periodic CAN data collection

(b) Device flashing

Fig. 5: Sequence diagrams for two selected functionalities

types should not make a big difference in the delivery of the data, the flashing process can indeed be different. The gateway in the vehicle needs to implement all the possible protocols used. Being such versatile while using limited resources makes it hard to design a generic solution.

C. Eco-driving

The goal of the function is to optimize driving habits through the collection of driving data (e.g., consumption, GPS data, measured forces, velocity, gear ratio, acceleration, etc.). Based on this driving data additional statistical cloud functionalities can be implemented to help drivers in achieving more economical driving habits. As there is no need for individual users to be able to monitor their incoming data from the vehicles sensors while they are driving or in quasi real-time, the system is designed so that it only receives the collected data every hour or after each trip if it was shorter than an hour. Based on these the system can create short reports from the data collected about the driver. This report evaluates the driver, mentions the major good and bad driving habits identified, the average values of fuel consumption, and highlighting the key areas where the user could improve their efficiency, also showing estimated savings. The user may generate more detailed reports on selected trips or time periods in addition to the evaluation.

D. Weather: report conditions and forecast

Based on several car related data a cloud connected community service could provide helpful information for drivers nearby about road conditions, weather, traffic and more. Such useful data sources are for example: wiper control state, fog light state, ABS, ESP data, inside/outside temperature and humidity sensors. The ground-up design of a complete and functional weather forecasting system is clearly out of the scope of this paper. Also, it doesnt seem very sensible to implement such system when every country has their own national weather forecasting service and furthermore there are organizations and companies that offer such services on a global scale. The best way to carry out this task would be to cooperate with such organizations or companies. The functionality concentrates on the *weather nowcasting* idea. Nowcasting means showing very short term weather predictions based on recent and localised measurements. Incorporating data collected from a large user group allows these predictions to be more precise and detailed than traditional weather forecasting. Given enough input data the system can provide information down to street-level resolution. The prototype function displays temperature values and rain on a map (see Fig. 6) via the backoffice application (see section IV-E). Additionally the maps can be extended with traffic information (average speed), accidents and congestions. The map should also warn about risks such as icy and slippery roads. A good example for a system like this is 511NY [22] which is a free service of the New York State Department of Transportation. The big difference between that system and ours is the source of data. While the 511NY service works based on sensors built into the road our system collects all the data from the vehicles. This means several advantages: no need to build and maintain costly sensors; the sensors are not fixed therefore can cover a larger area; changes in the road structure dont need attention to be integrated.

E. Backoffice

The Connected Car Backoffice System is a cloud-based web application intended to be used as a desktop interface to the different Connected Car functionalities, as the smartphone app should be used on the go. As an example Fig. 6 depicts the integrated user interface of the Weather functionality (see section IV-D). Backoffice itself provides the following functionalities for the system:

- **Reports**: All errors and warnings summarized for own cars, fleets and all manufacturers belonging to the user.

TABLE I: Results of load testing the 5 node Cassandra cluster with 90KByte payload/query using a 4VCPU/4GB RAM node

Thread count	Total ops	Adj. row/s	Row/s	Mean	Med	0.95	Max
4	74437	342	342	11.6	7.3	24.1	1631.4
8	218176	999	999	8.0	4	18.8	5392.7
16	411357	1963	1962	8.1	1.7	29.2	989.1
24	417118	1902	1902	12.6	1.6	50.3	2034.4
54	423090	1780	1779	30.3	1.7	142.5	962.9

This section is divided into the above mentioned 3 subsections. All of these show the number of faults for all the error and warning types. The parts required to repair the errors are also displayed. Two actions are available for the user here. They can request oil change for a car and can order parts for the car. The items can be ordered directly from a merchant partner or in case of a fleet, the collected orders can be stored, managed, printed or forwarded for internal or external use. When requesting an oil change the user can book a suitable appointment at a selected service station and the required oil type and amount are automatically forwarded.

- **Own cars**: All errors and warnings detailed for all cars owned by the user. The display is similar to the above section with additional information on oil change dates and mileages. However the structure is similar, the list here shows each individual car separately. The two actions of requesting oil change and ordering replacement parts are also available here for each car.
- **Fleets**: All errors and warnings for fleets administered by the user. The fleets are listed in the same fashion as in the summary section, but with more details. The two actions of requesting oil change and ordering replacement parts are also available here for each fleet. When ordering parts for a fleet all part required by all the cars in the fleet are added up with an option to exclude parts and cars from the list. Oil change appointments can be made with service partners or if the fleet has a private service station a private schedule can be managed internally.
- **Manufacturer**: All errors and warnings for manufacturers administered by the user. Car types can be checked for common issues; replacement parts can be tracked as well.

Additionally it provides a user and/or administrative interface for the different functionalities discussed in section IV. Functionalities utilize a role-based access control scheme. Each functionality has a defined set of roles that can access it. Partial access is also possible: different roles see different subset of data and functions for a functionality.

V. Evaluation

We deployed Variant 1 (see section III-A) on an OpenNebula [19] based IaaS cloud within SZTAKI. The deployment used the following resources: (i) 1 HAProxy node with 4 VCPUs and 2GB RAM; (ii) up to 5 Data Collector nodes with 4 VCPUs and 2GB RAM for each; and (iii) up to 5 Cassandra nodes each with 4 VCPUs, 4GB RAM and 60GB iSCSI storage from a storage area network (SAN). Additionally we used an additional node as Chef server; a local Amazon S3 compatible service (via Ceph RadosGW); and a node as MySQL instance. All nodes ran as virtual machines on AMD Opteron 6376 CPUs with all VCPUs mapped to actual physical cores.

For evaluating the variant we performed load testing with two scenarios: (i) Cassandra database cluster; and (ii) the whole data collector framework. For the first case we used the stress-testing tool (*cassandra-stress* [32]) provided by Cassandra. In the second case load tests were carried out with the Tsung load-testing framework [30] using the Zabbix [31] monitoring tool for observing the infrastructure during tests. The load testing framework includes 8 additional Tsung nodes with 4VCPUs and 2GB RAM each. The Tsung nodes were also deployed in the same OpenNebula, thus any measurement that involves network speed was always using the internal 10Gbit/s network.

A. Cassandra

First we used the *cassandra-stress* tool to determine the theoretical limit of number of inserts (row/s) for the database cluster using payload size estimation representing the *Weather nowcast* functionality of our platform (see section IV for more details). We used the 'SimpleStrategy' for replica placement and a replication factor of 1. We used a payload of 90KBytes of payload data per query. This is based on an internal estimation of the amount of data a single client (vehicle) will transmit after a data collection period of one hour. This consist of a fixed size header and the collected data.

Cassandra stress tool increases the number of threads writing to the database as long as there is no large decreases in write speed. This is denoted by "thread count. Table I shows a representative measurement. There total ops represent the rows written in this case, while row/s is the actual insert speed. Mean, median, 0.95 and max are latency related (we omitted 0.99, 0.999, time and stderr from the table). The tests were run always against an empty database, however the stress tool always inserts a large amount (150000-250000+, depends on the node count) rows as a warm up. Also it randomized the partition key to avoid hot spots in the cluster.

Our results show that the deployed system was able to sustain a median write speed of 2492.0 rows/s (with $\bar{x} = 2320.7$, $\sigma^2 = 431.80$) for thread count > 8.

B. Data Collector Framework

Second we load tested the whole data collector infrastructure as follows. We used different payloads of data representing a data collection interval between 1 and 60 minutes with payload data between 2.5 and 90 KBytes. We used a 8-node Tsung cluster for distributed load testing. Our goal was twofold: i.) to see which defined scenarios the framework/ architecture is able to complete; and ii.) how the performance compares to the performance measured in the raw Cassandra

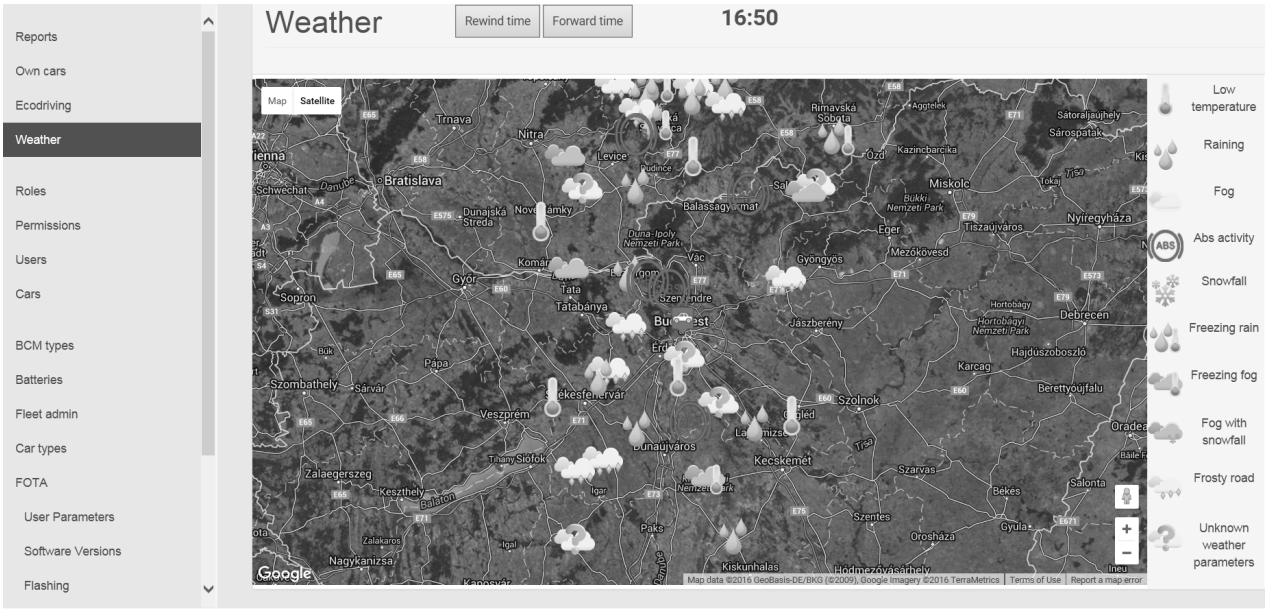


Fig. 6: Backoffice application with the user interface of ”Weather” (nowcast). The different functions and functionalities can be accessed through the menu on the left side.

load testing. We first ran 5 minute tests via Tsung and monitored via the HAProxy admin interface to see if any requests are denied and how many requests are queued up on the application servers. If the application servers were able to handle the load we repeated the test, but now 60 minutes long. The 60 minute tests was repeated 5 times. For the test we used a 5 node Cassandra cluster, up to 5 Data collectors and a single HAProxy.

Our results show that the simple deployment (with a single HAProxy and 5 Data Collectors) is capable of handling 360000 clients in a configuration of 2400 active users/sec with 2.5KBytes of payload, or 400 active/sec with 24KBytes of payload, or 100 active/sec with 90KBytes of payload data. This seems far from the raw Cassandra results, however the limiting factor is the processing power of the Data Collectors. As the results showed the number of active users can vary from 100 to 2400, depending on the payload data size. Fig 7 shows the transaction rate (requests/sec) and user arrrival and departure rate for a measurement representing 400 users/sec with 23.5KBytes of payload.

VI. Future work and conclusions

Vehicles are on their way of becoming the most sophisticated mobile devices in the world of the Internet of Things (IoT) with their vast amount of integrated sensors and on-board computers linked to the cloud by way of wireless technologies. On the other hand, there are various cloud technologies, deployment and service models already available. Therefore, creating a cloud-based IoT back-end for Connected Cars is not a straightforward task. In this paper we presented two well-established solutions and the major problems we faced during the development of such IoT platform for receiving, managing, distributing and visualizing vehicular data in various scenarios, including CAN data collection, device flashing, Eco-driving functionalities, and weather report/forecast. Two reference cloud implementations have been presented in details with benchmarks: an IaaS based system and an industrial PaaS based solution.

For future work, besides the previously described functions, we are working on implementing some other well-known functionalities to test the capabilities of the demonstrated platforms including the followings:

- *Remote Control functions* including keyless entry, remote engine start and remote climate control. A vehicle-user account system or a smartphone-vehicle wireless connection over Wi-Fi or GSM/LTE has to be developled here so that the vehicle and a driver/user can paired with each other. For the keyless entry function, which would allow the driver of a car to lock his/her vehicle from afar, a short range wireless connection like Bluetooth can also be an eligible solution. Several security issues arise when connecting to and from a smartphone device, the necessary precautions have to be made when establishing the pairing between the smartphone and the vehicle. To prevent unintentional or excessive use, some form of timing policy should be used/enforced on the usage of the remote control. For example, Tesla uses a 30 minute automatic shut off policy on their remote climate control application to prevent unnecessary battery drain. Similarly the user should be able to provide a time period when an extra security layer is in effect, like at a nighttime engine start or vehicle opening.
- *Car Sharing*: The envisioned car sharing system revolves around a cloud based user database, a smartphone application and a digital key or certificate, which is

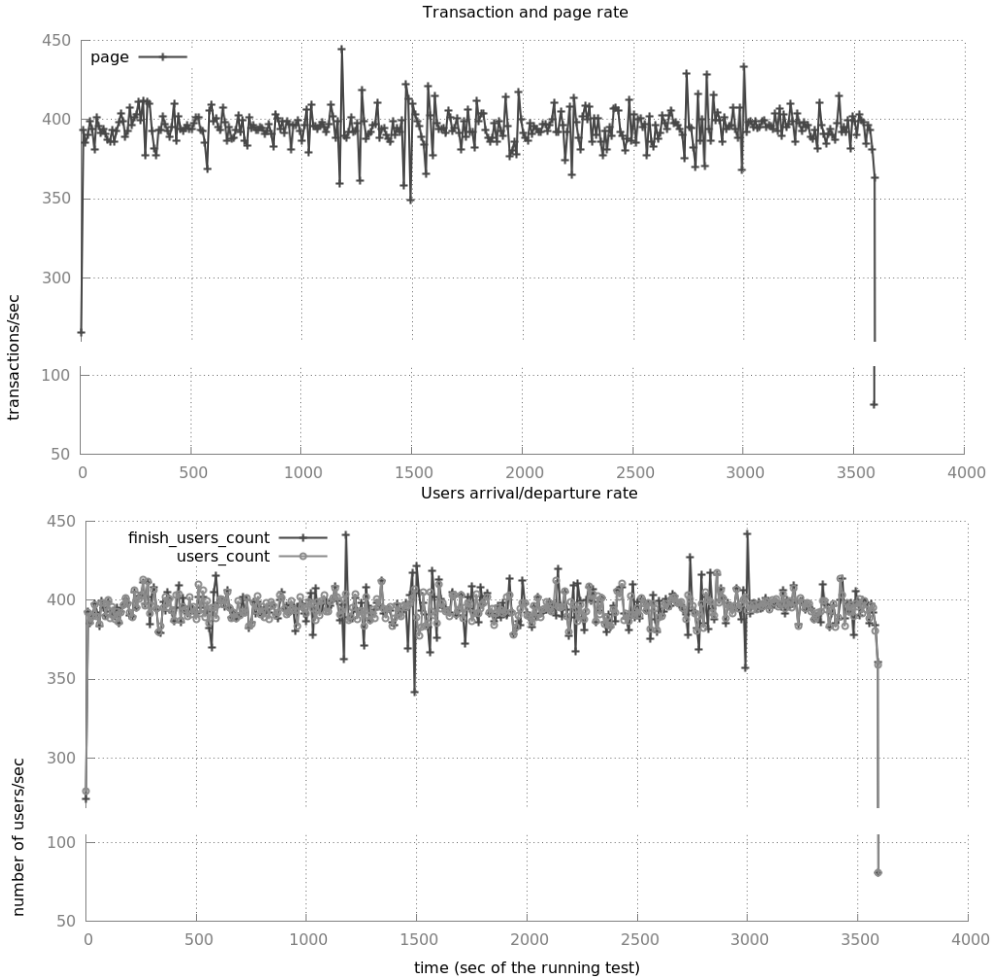


Fig. 7: Users arrival and departure rate; and Transaction and page rate for 400 users/sec and 23.5KBytes of payload

exchanged with the vehicle through a wireless network. The communication technology used can be anything ranging from GSM/LTE to Bluetooth or even NFC as it is presumed that the user will have some form of physical contact with the vehicle at some point, thus the shorter range of Bluetooth or NFC is not a problem. Once a user registered for a specific vehicle on his/her account, the server sends the forgery-proofed certificate in an encrypted format to the cell phone of the respective user. This certificate gets authenticated by the vehicle using its own certificate and once verified provides access to the vehicle. For this system to work properly multiple routes of communication are required. The vehicle needs to communicate with the owners phone and the third partys phone to which the vehicle was lent and the vehicle also needs to connect to the cloud. This can either be a direct GSM/LTE connection or a proxied connection over a smartphone.

- *Park pilot assistance*: This function tries to identify sufficient parallel or perpendicular parking space when passing them by. A smartphone displays this information, moreover should automatically look for a parking house nearby in case of unavailable parking spaces. This function can be extended with a community park spot finder feature. If enabled, this feature could collect information about available parking places found by the moving vehicle and transmit this information to the cloud service. When looking for a parking spot the application connects to the cloud service and downloads information about nearby parking places. In addition to the location of the spots the data contains the size and orientation of the spots. This way the system can consider size of the vehicle when searching. With this information the smart device can recommend a route with the highest probability to find the parking spot nearest to the destination.
- *Traffic assistant*: A group of functionalities that can be used for helping drivers avoid congestion on roads, warn them about traffic accidents or about required fill-up and accessory changes. With available and/or pre-recorded traffic sign geolocation data the validity of these signs can be verified.

REFERENCES

[1] Hberle, Tobias et. al. The Connected Car in the Cloud, A Platform for Prototyping Telematics Services. IEEE Software 32(6), pp. 11-17, November/December 2015

[2] He, Wu, et. al. Developing Vehicular Data Cloud Services in the IoT Environment. IEEE Transactions on Industrial Informatics 10(2), pp. 1587-1595, May 2014

[3] P. Mell, T. Grance, SP 800-145, "The NIST Definition of Cloud Computing, National Institute of Standards & Technology, Gaithersburg", MD, 2011

[4] F. Bajaber, R. Elshawi, O. Batarfi, A. Altalhi, A. Barnawi, S. Sakr, "Big Data 2.0 Processing Systems: Taxonomy and Open Challenges", Journal of Grid Computing 14(3), 379-405, 2016

[5] AWS Internet of Things. *https://aws.amazon.com/iot/* [Online] [Cited: 2017-10-30]

[6] Azure IoT SuiteIoT Cloud Solution *https://www.microsoft.com/en-us/internet-of-things/azure-iot-suite* [Online] [Cited: 2017-10-30]

[7] Google IoT Core. *https://cloud.google.com/iot-core/* [Online] [Cited: 2017-10-30]

[8] FIWARE Architecture Description: Big Data. *https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE.ArchitectureDescription.Data.BigData* [Online] [Cited: 2017-10-30]

[9] Marz, Nathan, and James Warren. Big Data: Principles and best practices of scalable realtime data systems. Greenwich, CT, USA : Manning Publications Co., 2015.

[10] In-stream Big Data Processing. HIGHLY SCALABLE BLOG. *https://highlyscalable.wordpress.com/2013/08/20/in-stream-big-data-processing/*. [Online] [Cited: 2017-10-30]

[11] Apache Hadoop.*http://hadoop.apache.org/* [Online] [Cited: 2017-10-30]

[12] The Apache Storm project. *http://storm.apache.org* [Online] [Cited: 2017-10-30]

[13] The Apache Spark project. *http://spark.apache.org* [Online] [Cited: 2017-10-30]

[14] The Apache Cassandra project. *http://cassandra.apache.org* [Online] [Cited: 2017-10-30]

[15] The Apache HBase project. *http://hbase.apache.org/* [Online] [Cited: 2017-10-30]

[16] Cloudera Impala. *http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html* [Online] [Cited: 2017-10-30]

[17] Yang, Fangjin and Merlino, Gian. Real-time Analytics with Open Source Technologies. SpeakerDeck. *https://speakerdeck.com/druidio/real-time-analytics-with-open-source-technologies-1* [Online] [Cited: 2017-10-30]

[18] FIWARE Glossary *https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE.Glossary.Global* [ONLINE] [Cited: 2017-10-30]

[19] R. Moreno-Vozmediano, R. S. Montero, I. M. Llorente, "IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures", IEEE Computer, vol. 45, 65-72, December 2012

[20] WebHDFS REST API. Hadoop. [Online] [Cited: 2017-10-30] *http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/WebHDFS.html*.

[21] Cascading: Application platform for enterprise big data. *http://www.cascading.org*. [Online] [Cited: 2017-10-30]

[22] New York Traffic. *https://511ny.org/* [Online] [Cited: 2017-10-30]

[23] HAProxy-the reliable, high-performance TCP/HTTP load balancer. *http://www.haproxy.org/* [Online] [Cited 2017-10-29].

[24] Gardner, James. "The web server gateway interface (wsgi)." The Definitive Guide to Pylons (2009): 369-388.

[25] The uWSGI project - uWSGI 2.0 documentation. *https://uwsgi-docs.readthedocs.io* [Online] [Cited: 2017-10-30]

[26] Reese, Will. "Nginx: the high-performance web server and reverse proxy." Linux Journal 2008.173 (2008): 2.

[27] *https://www.cloudfoundry.org/* [Online] [Cited: 2017-10-30]

[28] Weil, Sage A., et al. "Ceph: A scalable, high-performance distributed file system." Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 2006.

[29] Chef: Deploy new code faster and more frequently. Automate infrastructure and applications. *http://chef.io* [Online] [Cited: 2017-10-30]

[30] Tsung: a distributed load testing tool. *http://tsung.erlang-projects.org/* [Online] [Cited: 2017-10-30]

[31] Tader, Paul. "Server monitoring with Zabbix." Linux Journal 2010(195), Article No 7, July 2010

[32] The Cassandra-stress tool. *http://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsCStress.html* [Online] [Cited: 2017-10-30]