Intelligent Services for Big Data Science

C. Dobre[a,1,*], F. Xhafa[b]

[a] *University Politehnica of Bucharest, Splaiul Independentei 313, Bucharest, Romania*
[b] *Universitat Politecnica de Catalunya, Girona Salgado 1-3, 08034 Barcelona, Spain*

Abstract

Cities are areas where big data is having a real impact. Town planners and administration bodies just need the right tools at their fingertips to consume all the data points that a town or city generate and then be able to turn that into actions that improve peoples lives. In this case big data is definitely a phenomenon that has a direct impact on the quality of life for those of us that choose to live in a town or city. Smart Cities of tomorrow will rely not only on sensors within the city infrastructure, but also on a large number of devices that will willingly sense and integrate their data into technological platforms used for introspection into the habits and situations of individuals and city-large communities. Predictions say that cities will generate over 4.1 terabytes per day per square kilometer of urbanized land area by 2016. Handling efficiently such amounts of data already is already a challenge. In this paper we present our solutions designed to support next-generation Big Data applications. We first present CAPIM, a platform designed to automate the process of collecting and aggregating context information on a large scale. It integrates services designed to collect context data (location, users profile and characteristics, as well as the environment). We next present a concrete implementation of an Intelligent Transportation System designed on top of CAPIM. The application is designed to assist users and city officials better understand traffic problems in large cities. Finally, we present a solution to handle efficient storage of context data on a large scale. The combination of these services provide support for intelligent Smart City applications, for

*Corresponding author.
Email addresses: `ciprian.dobre@cs.pub.ro` (C. Dobre), `fatos@lsi.upc.edu` (F. Xhafa)

[1]This work was supported by project "ERRIC -Empowering Romanian Research on Intelligent Information Technologies/FP7-REGPOT-2010-1", ID: 264207.

actively and autonomously adaptation and smart provision of services and content, using the advantages of contextual information.

Keywords: intelligent services, big data, context-awareness, mobile computing

1. Introduction

Every day we create 2.5 quintillion bytes of data; so much that 90% of the data in the world today has been created in the last two years alone. This data comes from sensors used to gather climate information, from posts to social media sites, digital pictures and videos, purchase transaction records, or cell phone GPS signals, to name only a few. This data is *big data*. Analyzing large data sets already underpins new waves of productivity growth, innovation, and consumer surplus. Big data is more than simply a matter of size; it is an opportunity to find insights in new and emerging types of data and content, to make businesses more agile, and to answer questions that were previously considered beyond our reach. Until now, there was no practical way to harvest this opportunity. But today we are witnessing an exponential growth in the volume and detail of data captured by enterprises, the rise of multimedia, social media, and the Internet of Things.

Many of Big Data challenges are generated by future applications where users and machines will need to collaborate in intelligent ways together. In the near future information will be available all around us, and will be served in the most convinient way - we will be notified automatically when a congestion occurs and the car will be able to decide how to optimize our driving route, the fridge will notify us when we the milk supply is out, etc. Technology becomes more and more part of our daily life. New technologies have finally reached a stage of development in which they can significantly improve the lives of any city's inhabitants. Our cities are fast transforming into artificial ecosystems of interconnected, interdependent intelligent digital "organisms". They are transforming into *smart cities*, as they benefit more and more from intelligent applications designed to drive a sustainable economic development and an incubator of innovation and transformation that merges the virtual world of Mobile Services, Internet of Things and Social Networks with the physical infrastructures of Smart Building, Smart Utilities (i.e. electricity, heating, water, waste, transportation, and unified communication & collaboration infrastructure). The transformation of the metropolitan land-

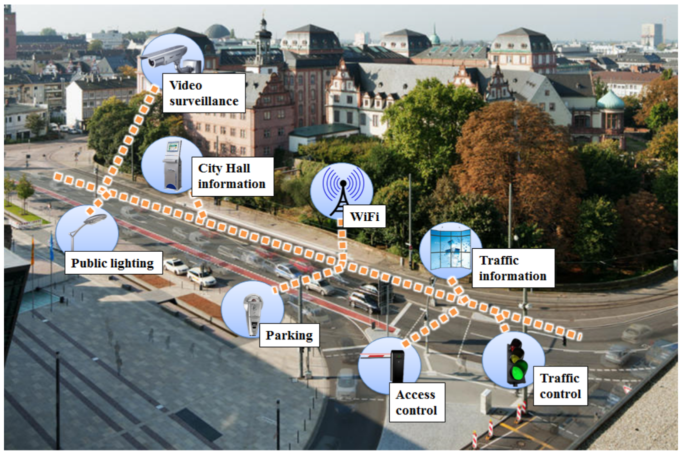


Figure 1: The vision of integrated community-enabled intelligent services.

scape is driven by the opportunity to embed intelligence into any component of our towns and connect them in real-time, merging together physical world of objects, humans and virtual conversation and transactions.

There are already examples of Smart Cities like Malaga, Amsterdam, or Boston. In Europe projects such as "European Initiative on Smart Cities" (SETIS) ignited many local Public administrations to launch new initiatives to take advantage of the opportunity of funding for Smart Cities [1]. Several ICT companies (e.g. IBM: Smart Planet, Accenture: Intelligent City Network, CISCO: Connected Urban Development, Ericsson Smart City, etc.) and research institutions (e.g. MIT: Smart City SENSEable lab, Terreform One, etc.) already offer services and solutions' components that can help to build more livable, sustainable cities by innovative ICT usage. In many towns such as Boulder or Amsterdam many ICT companies are working together with utilities, universities and other organization to provide integrated solutions. In 2016, it is estimated that \$39.5 billion will be spent on smart city technology, up from \$8.1 billion in 2010 [2].

We came a long way since the original "wired city" vision advocated in the 1980s [3]. The notion of wiring the city then was needed to support networking very diverse activities and routine services such as those provided by municipalities - libraries, welfare services - over WANs. Today, the development of ubiquitous devices of comparatively low cost that can be deployed to sense what is happening over very small time scales, as well as over very fine levels of spatial resolution. Such devices that range from purpose-built sensors to individual hand-held devices that are as mobile as those using them provide massive capability to store and transmit data that pertains to movement and activity levels across space and time. Some of the most elaborate applications involve transport, but other services can easily be integrated all-together as well (see Figure 1).

However, today there is no unique model for a Smart City, and each city approaches the concept in its own particular way, with different projects and objectives. One of the obvious but much misunderstood features of these new urban technologies is the fact that *they produce massive streams of data in real time and space*. We are just beginning to grasp the nature of this 'big data'. So far, most of the datasets from which scientists and researchers were able to extract real meaning are quite small in comparison with the sort of data that can be captured by smart city applications. Imagine the movements of people in a large city like London, for example, where there are something like 3 million travellers a day using some form of public transport. If the municipality could capture data about locations of these travellers to extract meaningful information, with technology available today probably much of this data can be reduced or aggregated. Thus, sequences with evident meaning could be in extracted. But things become more complicated if we think that such kind of data is available continuously. Over sufficiently long periods of time, one can begin to extract changes to the structure and form of the city and the way people behave. But this yields for solutions to store and manage tremendous amounts of data. And all this data will probably need to be supplemented with all kind of other information (e.g., transport data, data relating to social and economic interactions).

We have barely begun to get a sense of the dimensions of this kind of data, of the privacy implications, of ways in which we can code it with respect to meaningful attributes in space and time. As we move into an era of unprecedented volumes of data and computing power, the benefits aren't for business alone. Data can help citizens access government, hold it accountable and build new services to help themselves. In one sense, all this is part of a world that is fast becoming digital in all its dimensions, where we can develop our understanding and our design ideas digitally, using representations and data that are also digital and developing new ideas for the future which will be implemented and will change the digital basis of everyday urban and social life.

In this paper we present our solutions designed to support next-generation Big Data applications. The contribution of this paper is as follows: We present first CAPIM, a platform designed to automate the process of collecting and aggregating context information on a large scale. It integrates services designed to collect context data (location, users profile and characteristics, as well as the environment). These smart services are dynamically loaded by mobile clients, and make use of the sensing capabilities provided by modern smartphones, possibly augmented with external sensors. The data is collected and aggregated into context instance. This is also possible augmented with external and inferred data about possible situations, relations, or other events.

The platform is specifically designed for collaborative environments where people collect and share data to support the understanding of their surroundings. For example, people can use their smartphones or tabletPCs to collect information about traffic (congestions, pollution, or safety-related events), and we present next as a case study a concrete implementation of such an Intelligent Transportation System designed on top of CAPIM. The application is designed to assist users and city officials better understand traffic problems in large cities.

Finally, we present our solution, called Context Aware Framework, to deal with efficient storage of context data on a large scale, on top of a popular Cloud storage system called BlobSeer. Context Aware Framework automates the provisioning of context data, and sits between a persistence layer, where data is actually stored in the Cloud environment, and the actual context-aware application (CAPIM), running on the users mobile devices. In our vision, a truly context-aware system is one that actively and autonomously adapts and provides the appropriate services or content to the users, using the advantages of contextual information without too much user interaction. Thus, providing efficient mechanisms for provisioning context-sensitive data to users is an important challenge for these systems. Context Aware Framework is designed to support the storage of context data for such context-aware systems. It manages every problem related to, for example, the unpredictable wireless network connectivity and data privacy concerns over the network, providing transparent access to the data to such systems.

The work extends on our previous work [4] with extensions designed for handling Big Data, as well as with a concrete implementation of a Smart City application that uses the CAPIM's capabilities.

The rest of this paper is organized as follows. Section 2 presents Related Work. Section 3 presents an analysis of Big Data challenges and requirements coming from Smart City applications. We next present a platform designed to support automatic acquisition and handling of context data, specifically designed in support for such applications. As a case study, Section 5 presents an application designed for traffic handling in large cities. In Section 6 we present the solution designed in such applications to handle the efficient storage of large amounts of contextual information. Section 7 concludes the paper.

2. Related Work

The ubiquity of mobile devices and sensor pervasiveness, e.g., as in smart city initiatives, call for scalable computing platforms to store and process the vast amounts of the generated streamed data. Because of the complexity in dealing with large amounts of data, several platforms were previously developed to service Smart City applications and hide the complexities related to how the context information is gathered, stored and processed. This way, the Smart City developer is left only with the implementation of the necessary business logic, which can use the already-serviced high-level situation information.

Several architectures and frameworks have been proposed in order to support the development of context-aware systems. One of the first implemented approaches is the Context Toolkit [5]. The framework presents an architecture composed of different functional modules in order to acquire, aggregate and interpret context information. It uses (key/value) pairs in order to model context data. Other approaches like CASS [6] propose a layered middleware architecture that uses a relational data model to represent context data. JCAF [7] is a framework and a runtime environment to develop and deploy contextual computing applications. It uses an object oriented model to represent context data. These three approaches use interpreters to convert acquired raw data into higher level context data, but these transformations cannot be very complex because there is no inference mechanism.

The CoBra [8] middleware proposes a different approach, where software agents are used in order to acquire and process context data in a smart meeting room environment. SOCAM [9] uses three different layers, namely a sensing layer, a middleware layer and an application layer. Still, these middleware are designed to work only with limited collections of data. The context is discussed in only few works in relation with Cloud Computing. For example, Boloor et al [10] studied the dynamic request allocation and scheduling for context aware applications in geographically distributed data centers. But, to our knowledge, treating the specific issues and requirements of context-awareness for Big Data storage is practically missing in actual works.

Cloud computing provides some of the features needed for these massive data streaming applications. For example, the dynamic allocation of resources on an as-needed basis addresses the variability in sensor and location data distributions over time. According to the Association for Computer Operations Management (AFCOM), in 2011 90.9 percent of data center sites used more storage space than they did three years ago. During the same three-year period, 37 percent were able to reduce their staff, and 29 percent kept their staffing levels the same. This trend is in large part due to the development and implementation of new tools and processes that have allowed IT departments and data centers to store massive amounts of data efficiently and inexpensively. The advent of cloud-based storage systems has had since then a profound impact on the way businesses collect and store their information. However, todays cloud computing platforms lack very important features that are necessary in order to support the massive amounts of data streams envisioned by the massive and ubiquitous dissemination of sensors and mobile devices of all sorts in smart-city-scale applications.

For massive context data streaming applications, M3 [11] is a prototype data streaming system that is being realized at Purdue using Hadoop. M3 eliminates all of Hadoop's disk layers, including the distributed file system (HDFS), and the disk-based communication layer between the mappers and the reducers. It proposes a hybrid memory-based and disk-based processing layer, includes dynamic rate-based load-balancing and multi-stream partitioning algorithms, and fault-tolerance techniques. However, M3 can handle only streaming data and does not handle queries that mix streaming with disk-based data. A context awareness extensible layer for M3 has been demonstrated separately in Chameleon [12]. However, Chameleon lacks general context-based indexing techniques for realizing context awareness, thus when the context changes the system cannot easily augment the query being executed by additional predicates to reflect that change.

Similar to CAPIM, [13] presents a large scale system, called Federated Brokers, for context-aware applications. In order to avoid the centralized design(single point of failure) found in previous papers, the authors propose a context-aware platform that includes multiple brokers. In addition to this, CAPIM (and the Context Aware Framework, its component designed for data storage) uses also a metadata manager for all stored information (which relieves much of the burden impose for managing data on the mobile application). We also offer prediction capabilities based on the provisioned data. Also notable is that the evaluation of Federated Brokers was conducted over a small-size homogeneous environment. We actually tested Context Aware Framework over a large grid environment, with more brokers and clients, considering a large distributed storage configuration.

From an utility point of view, CAPIM can also be compared with Google Now [14] and Microsoft On{X} [15]. Google Now [14] is able to predict what information an user need, based on his previous searches and on his context data. Microsoft On{X} lets the user set actions for states defined by his context data. When a certain state (previous defined by the user) is reached, a trigger is fired. Our platform supports this kind of approaches, being build as a framework for developers, not as a stand alone application, like Google Now or Microsoft On{X}.

These and similar other middlewares support differently pervasive and mobile computing based on context information. They all provide some method of adapting to changes in the context, and methods for collecting context, but otherwise use different entities and have different focus. We present a more complete and complex context model that integrates a wider spectrum of information, ranging from location to users profile and social capabilities. The middleware allows collecting context information from a wide-area of data sources, aggregation including providing semantic relations, and an engine that is able to mimic the behavior of various context-aware applications.

3. Big Data challenges and requirements for Smart City applications

Smart City applications are highly dependent of their execution context. The term 'context' itself was considered by different authors as either the surroundings of the interaction between the user and the application [16], the information about the activity or the task the user is currently performing [17], or the needed information to characterise the situation of a given entity [5]. More generically, context is considered as any information that can be obtained and processed by a system to identify the situation of an entity (person, place or object), and adapt the system's behavior to that situation. It can be the GPS signal monitored using the user's smartphones, from which the system can infer the user's current location.

A context-aware platform designed to handle context information on a large scale should deal with several specific requirements:

- *Mobility and locality.* A context-aware application can help users augment their reality. In this case, the user might be interested to receive information about neighbouring places or buildings (e.g., in a tourism application). We assume that users are generally *moving*, and typical context data includes elements such as current locality, time, and user's status.

- *Proximity* is also important for provisioning. The amount of data is potentially too large to be served entirely on the user's mobile device, thus a selection of only the most relevant context data, from the immediate surrounding environment, is preferred.

- *Real-time guarantees.* Context-aware applications should provide *real-time guarantees* for data provisioning. The user should not receive events that happened too far in the past (if events are too old they become obsolete). For example, if a tourist is looking for information about objectives in his surroundings, he will not be very happy receiving data for things in the other part of the town simply because the application is answering requests made some while ago.

- it Support for communication imperfection. We acknowledge the imperfections of today's wireless communication infrastructures. No application should, in fact, assume that user is always connected to Internet (a wireless connection might not always be available, or in roaming the connection might not come cheap). Thus, the context-aware paltform should *support the use of context data even when an Internet connection is not available.* Alternatives such as opportunistically using the data accessed by others from distributed caches using only short-range communication (in form of Bluetooth and/or WiFi) can be used in such situations.

- *Efficient data access.* An application should allow *efficient access to the data* - in terms of speed of access, as well as support for complex queries. Applications should be able to express their interests using complex queries, in forms of naturally-expressed filters. For example, the application should be able to request the data using an expression similar to “get prediction of my friends’ location, but only for those in town”. Or, an aggregated request could be expressed as “get prediction of road traffic on a particular street”.

- *Support for efficient data storage.* Context-aware platform should provide *discovery and registration of data sources* (e.g., sensors and external services such as a weather service), access to data using different granularities, and the aggregation of information. They also need to be *scalable.* For a typical collaborative traffic application, the number of users could potentially be in the range of millions. The sensed data should further be *persistently stored*; the history of data should be preserved for traceability and advanced data mining processing.

In summary, data intensive context-aware applications have common requirements concerning the high data volumes and fast access to data. Such requirements have to be satisfied by the services provided at the platform level, where the complexities related to handling context data should be shielded from the Smart City application. Obviously, achieving highly scalable data management is a critical challenge, as the overall application performance is highly dependent on the properties of the data management service. Cloud computing responds to this challenge by its computing and cost models: on-demand use of resources, pay-as-you-go pricing model, offering resources (CPU, storage) as utilities, etc. The highly scalable storage stays at the base of the “infinite capacity” slogan used for attracting customers. Users can reserve capacities in advance and release them when they are no longer needed.

On the other hand, supporting a large number of concurrent accesses and the interoperability with various (mobile) devices via Internet connections respond to the needs of modern interactive applications. In this respect, Clouds use different access methods and corresponding APIs (Web service, file based, block based, etc.), improved transport protocols (e.g. the Fast and Secure Protocol - FASPTM), and distribution of resources, including the geographic distribution of data storage resources [18].

Context-aware applications have additional requirements and a special profile, since context data can be used not only to accurately understand the semantics of business data in the benefit of applications, but it can be exploited for improving the performance and facilitate the management of Big Data store services. Due to the heterogeneity of data sources (from sensors to users and applications), various types of context data, in different formats must be persistently stored and offered to collaborative applications. Also, the guarantee of real-time exchange of data is required to address the reactiveness of context-aware applications. These various storage, transformation, delivery or archiving requirements make the task of context data management very complex, and claim for new architectural features and functionalities to be added to the data storage.

4. Case study: CAPIM, a platform to support intelligent context sensing

CAPIM (Context-Aware Platform using Integrated Mobile services) is a monitoring platform [4] that integrates services designed to collect context data (location, user's profile and characteristics, as well as the environment). Such smart services make use of the sensing capabilities of modern smartphones and tabletPCs, possibly augmented with external sensors. CAPIM's architecture consists of several layers (see Figure 2), each one providing a specific function: 1) collecting context information, 2) storing and aggregation of context information, 3) construction of context-aware execution rules, and 4) visualization and user interaction. All layers are composed of several components, making the platform suitable for experimenting with a wide range of context-aware methods, techniques, algorithms or technologies. It can be used to construct context-aware applications using a service-oriented composition approach: load the core container, instruct it to load the necessary context-gathering services, deploy a corresponding context-aware business workflow and call the actions to be executed when context is met.

First the user installs on his smartphone or tabletPC the platform container. This is the execution framework on which all layers are built. The monitoring services are dynamically discovered, downloaded as needed, loaded and executed inside this container. So, for collecting context information, the **first layer** includes sets of monitoring services (collecting and first-stage storing on the local mobile device) for the context data. Each monitoring service is packed in a digitally signed monitoring module. These modules are downloadable from remote repositories, resembling application stores. The monitoring services can be developed/maintained by third party organizations (for example, a manufacturer might construct a module to collect data from its own sensor, therefore integrating its data within the user's context). Each monitoring service is also executed inside a separate container. This

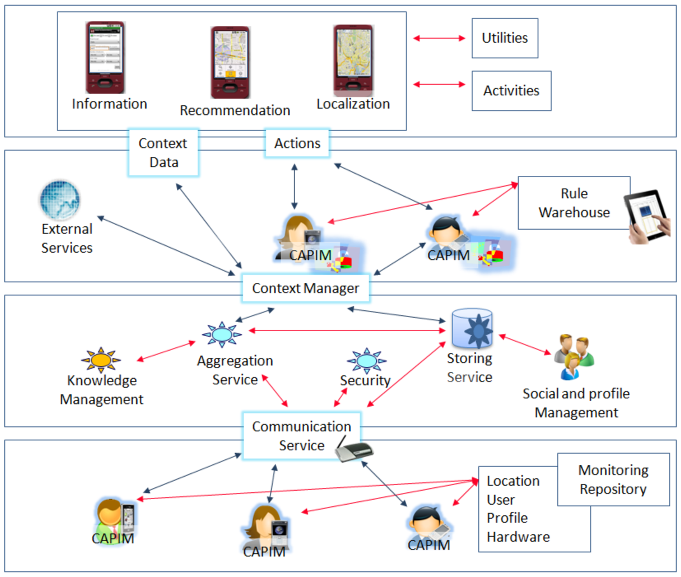


Figure 2: CAPIM's architecture.

allows separation of concerns (no service needs to know what other modules are deployed) and fault isolation.

The monitoring flow is under the control of a Context Manager (see Figure 3), orchestrating the flow of information between the monitoring services. Depending on the function supported, the monitoring services are grouped in several categories. The Push and Pull monitoring services are directly responsible for collecting context information. They collect context information generally directly from sensors. The Push service reacts to changes of the context, which in turn triggers notifications to be sent to the Context Manager. The Pull service is periodically or on-request interrogated for the current values of the monitoring parameters.

The context information is sent to Filter, Storage and Networking services. The Filter service subscribes to specific context information. The Context Manager forwards the data of interest to the Filter service, which in turns can produce new context information (possible from multiple data sources). Such a construction allows for first-stage aggregation of context information. The Storage service can store data locally for better serving the context-execution rules. Finally, the Networking service is responsible for sending the collected context information remotely to aggregation services (the Remote Context Repository component located in the next layer). Here

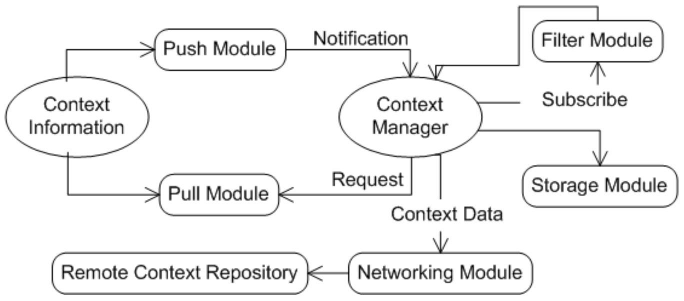


Figure 3: Flow of monitoring information.

we can experiment with different network protocols and methods of sending data, whilst balancing between costs and energy-consumption.

The second layer deals with the aggregation and storing of context data. These components are running on the server (in Cloud) - the data aggregation involves collecting information from multiple mobile sources (users running around and sensing data within the city send their data here) and sending it to the server-side aggregation service, where it is semantically organized. Thus, at this layer the information is aggregated from several context sources. The data from several sensors (GSM, WiFi, Bluetooth) is, for example, aggregated into current Location. CAPIM uses for that a semantic model[2].

For semantic aggregation, CAPIM uses the Jena Semantic Web Toolkit [19]. The framework provides functions to add, remove, query even to infer data on generic RDF models. CAPIM uses a semnatic network (see Figure 4). This context ontology can capture context information and model the basic concepts of person, interests and activities, describing relationships between such entities. The ontology is composed of domain-specific ontologies: The FOAF [20] ontology is used to describe user activities and relations with other people and objects . To describe events, dates or locations we use the ICAL and GEO [21] / WAIL [22] ontologies. Thus, we use common vocabulary to manage and share context data. The advantage of such an approach is sharing a common understanding of information to users, devices and services, because the ontology includes machine-interpretable definitions of basic concepts and relations. Using domain-specific ontologies we can dynamically

[2]However, other models are supported as well. For example, data can also be collected as time series, for long-term and near real-time processing guarantees.

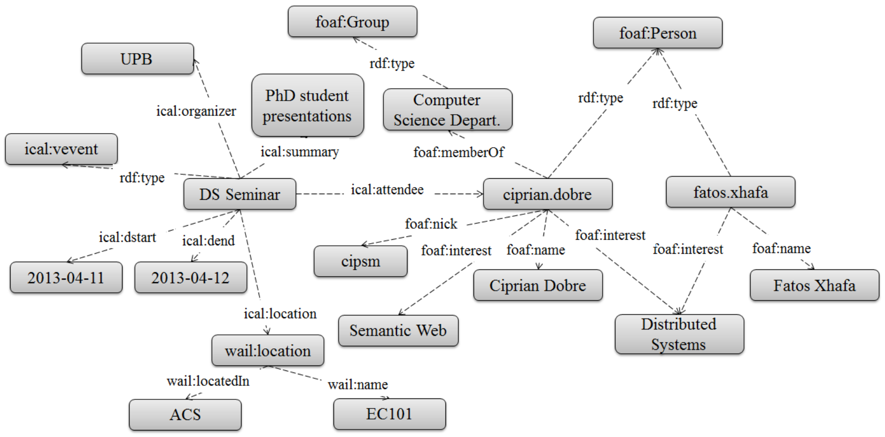


Figure 4: Part of CAPIM's Semantic Network.

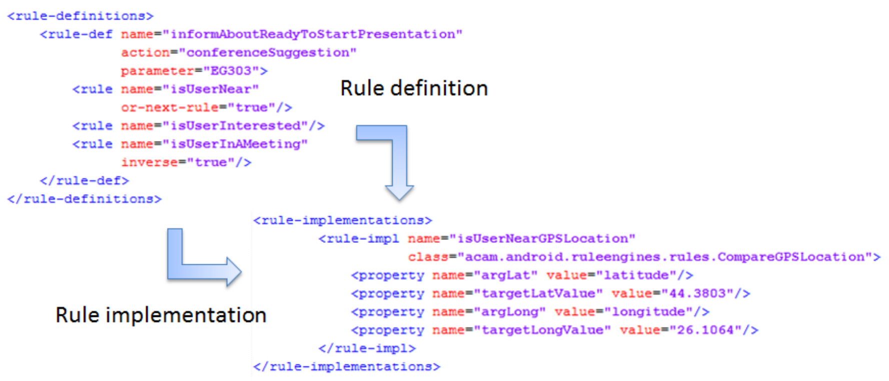
plug and unplug them from our model when the environment has changed. Also because the CAPIM's ontology is based on already implemented ontologies, the redundancy can be avoided and the semantic stored data can be easier linked with other information on the web.

Next, CAPIM is able to automate the execution of context-dependent actions. Changes in the context can trigger actions on the mobile phone according to a predefined rule set. The rules are expressed in an XML-based format (these are either community-built rules that can be remotely downloaded, or the user can contruct his own rules).

First, the context information is translated into a list of parameters (*key-value* pairs) that are used for defining the conditions inside the rules configuration file.

The rule definition contains a list of rule elements that are periodically evaluated by the engine. A rule is composed of several elements: *Conditions* (expressed as Boolean expressions, based on rule implementations), *Actions* (an action is triggered when the rule conditions are met), and *Action* parameters (strings which are passed as parameters to the action).

The rule implementations specify different expressions used to evaluate the context. For example, *rules.StringFieldEquals* expresses equality between a context parameter and a string. This rule can be applied to context parameters which have string values, such as the user's name. By default CAPIM provides several such rule implementations: *StringFieldEquals*, *IntFieldEquals*, *IntFieldGreaterThan*, *IntFieldLessThan*, *IntFieldBetween*, etc.



```
<rule-definitions>
    <rule-def name="informAboutReadyToStartPresentation"
              action="conferenceSuggestion"
              parameter="EG303">
        <rule name="isUserNear"
              or-next-rule="true"/>
        <rule name="isUserInterested"/>
        <rule name="isUserInAMeeting"
              inverse="true"/>
    </rule-def>
</rule-definitions>
```

```
<rule-implementations>
        <rule-impl name="isUserNearGPSLocation"
                   class="acam.android.ruleengines.rules.CompareGPSLocation">
            <property name="argLat" value="latitude"/>
            <property name="targetLatValue" value="44.3803"/>
            <property name="argLong" value="longitude"/>
            <property name="targetLongValue" value="26.1064"/>
        </rule-impl>
</rule-implementations>
```

Figure 5: Example of rule definition and implementation.

The user can also specify higher-level functions: he can combine rules using boolean algebra, or he can implement custom-made aggregation functions. In the second case, the data is first passed to that component, and the result is further used in the rule evaluation.

The rules represent operations between basic types, and allow one to formulate different restrictions on context parameter values. When combined they can lead to more complex conditions:

$$Rule = Rule\ OR\ Rule \mid Rule\ AND\ Rule \mid RuleImpl$$

The rule XML file is structured in two parts: *rule-definitions* and *rule-implementations* (see Figure 5). The rules specified in the *rule-definitions* part under the *rule-def* tag are complex rules, the ones that trigger a certain action if they are evaluated to true. Complex rules are described with a *name* attribute, an attribute to specify the action to trigger and possibly an attribute for the parameter of the action.

In defining a complex condition as a list of simple rules, if the *or-next-rule* attribute of a simple rule is not specified as having the value “true”, then the current rule is in an *logical AND* relation with the next rule. On the other hand, simple rules may have the inverse attribute set to true such that we obtain a negation of the rule. Taking this into consideration, the rule in Figure 5 can be translated to the following logical proposition:

```
1  if $user\ is\ near$ OR $user\ is\ interested\ in\ the\ subject$
2   $of\ the\ presentation$ AND (NOT $user\ is\ in\ meeting$)
3      then $inform\ user\ about\ presentation$
```

Finally, the fourth layer is responsible with the applications, expressed as rules and actions, which can be used for orientation, information and recommendation purposes. At this layer there are local utilities that can help with context-triggered actions. Also applications can use the context data to improve response to stimulus (an interior or exterior request). An application can react to changes in the current context and take specific actions depending on some predefined rules. For this, conditions are evaluated period as the data is retrieved. Third party applications and services can use the API provided by the context-aware services. They can use functions for obtaining particular context data, using filters, or can subscribe for context data. They can also declare new execution rules for users to install on their mobile devices.

4.1. Analysis of requirements and lesson learned from CAPIM to support Smart City applications

By developing CAPIM, we encountered several challenges related to efficiency of storing large volumes of sensed data. These challenges were further incorporated into the intelligent storage system presented further in Section 6.

In terms of data accesses, in CAPIM *users write frequently*, while they *read the data in a sparse way.* They have also an interest in storage of large data volumes, for mining and processing relevant high-level context information. Such requirements are generally encoutered in many other Smart City applications. To cope with them, for persistance, collaborative and mobility support, we designed the layered architecture: the data is first stored locally, on the mobile device itself, for a short period of time. It is also stored with different granularities on the server-side (the first aggregation layer). Finally, for persistence, data is further stored for long-term on a Cloud storage support.

For persistance storage, a typical relational database has the disadvantage that accesses to the same data units should be synchronized for strong consistency guarantees. This cannot support well a typical scenario envisioning millions of concurrent users writing their context data. Thus, we turned our attention to alternative solutions. BlobSeer [23] is a large-scale, distributed, binary storage service. It keeps versions for all records, so that concurrent read/write accesses are facilitated without affecting the high throughput of the system. BlobSeer allows concurrent accesses to the data, and for that is uses a versioning mechanism. Another benefit is that BlobSeer allows fine grain access to the data: it is possible to access small chunks, without having to read the entire Blob for example. BlobSeer also offers high throughput for read and write operations: clients can write new information in a chunk while others can read the old information, without needing to synchronize. Thus, BlobSeer [23] offers an appropriate alternative, as it provides real-time guarantees, large concurrent access guarantees, and support for eventual consistency through an advanced versioning mechanism.

5. Case study: an Intelligent Transportation System

We present a real-world application that uses the benefits of CAPIM. Applications designed for Intelligent Transportation have the potential to solve problems related to traffic congestions, by support a more efficient use of existing road networks, resulting in reduced traffic congestion, delays, emissions, energy consumption and improved safety. The success of City Hall strategies for transportation depends, to a large extent, on the quality and accuracy of the information given to the drivers. Technologies related to Intelligent Transportation Systems (ITS), with advanced traveller information systems (ATIS), travel advisories, variable message signs (VMS) and others, attempt to relieve congestion and decrease travel time by assisting drivers on selecting routes, departure times, and even mode of travel. Also, such technologies rely on a large extent on collective and computational intelligence - which is why CAPIM was selected as an ideal candidate [24].

The service, called *Traffic Collector* , was designed as an extension of CAPIM to gather information about daily traffic. Traffic Collector is installed on smartphones, tabletPCs, or computers inside cars, to collect traffic data. Unlike previous solutions that rely on sensors placed within the road infrastructure, here the drivers help sensing the traffic: the application automatically collects data about speeds, travelling conditions, etc. Such a crowd-based approach can produce a more accurate model of the traffic inside a city, and does not require the use of additional sensing devices. In return, to motivate users collect data, we rely on several crowd-sensing incentives: the driver receives feedback to optimize his cruising experience, he can obtain personalized driving statistics. We also introduce a social driver experience, where the driver more easily interacts with his environment and people around.

Traffic Collector captures data collected from sensors, stores it locally on the mobile device, and uploads it when an WiFi connection becomes available (but it can also send data in real-time over 3G, WiMAX and other protocols). For Traffic Collector first we had to insert mechanisms to get a critical mass of users to give up parts of its (limited) computation and energy power and run the application while in traffic. So, incentives are very important to sustain such collective data sensing technologies.

The application is intended for users that drive their cars around the city. It targets users that drive every day and do not have a fixed pattern (from work to home and back or from school to work and back etc.) like lorry drivers, taxi drivers and others. However, we also consider bus drivers, as they can provide accurate data concerning the driving conditions.

Getting a critical mass of users to participate and collect was a challenge. Kauffmann and Schulze [25] suggest that there are intrinsic and extrinsic motivations that cause people to contribute to crowd sourcing solutions. Intrinsic motivations refer to enjoyment-based (related to fun and enjoinment that the contributor experiences) and community-based motivations (refer to community participation and include community identification and social contact). The extrinsic motivations are broken into three parts: immediate payoffs, delayed payoffs and social motivations. Since payoffs (gaining money from offering services) are not considered, we turned our attention towards the implications of social incentives, as altruistic motivations for sensing within a given community.

Based on the initial feedback from some of our (student) users, we first integrated into Traffic Collector the ability for the user to keep a detailed log of his travels. Therefore, the user is presented with with different statistics about the traveling speeds on different roads, average traveling times, itinerary information, etc. This is done using Google Maps API, on top of which we placed different markers to color-code speeds (see Figure 6). Next we added more visual information, such as a custom gause capable to present the current traveling speed, the current altitude, or the distance covered by the driver. The user can also use statistics related to the maximum speed, the average speed, etc. The third feature added was for people to enter social networking information (name, Facebook username and Twitter username). The idea was to let the driver see the position of his/her (Facebook or other online social networks) friends, and post text related to his/her current position on Facebook or Twitter.

The application has been released to Android Google Play [26], and we had over 100 installs in one month from its publishing.

In addition to the Traffic Collector mobile application (the one using

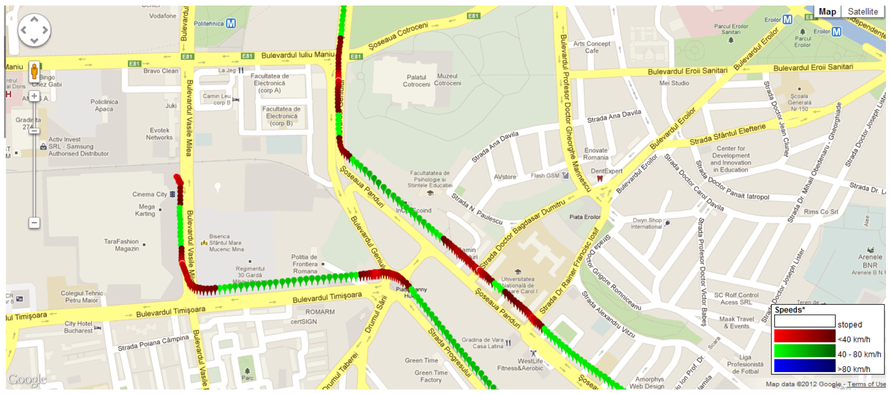
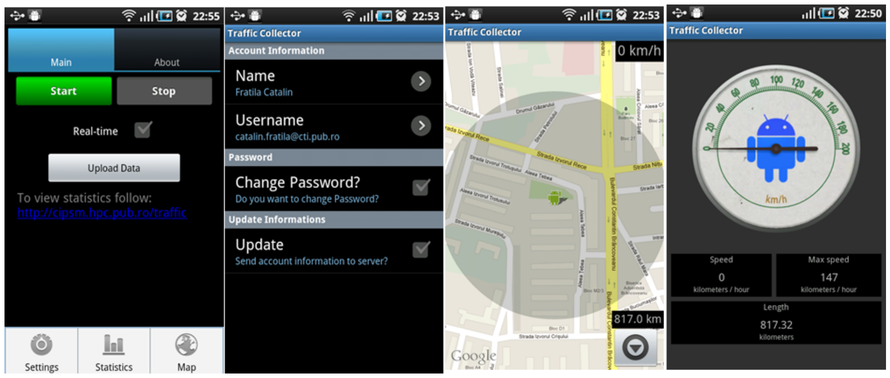


Figure 6: Speed statistics presented to the user/driver.



To view statistics follow:
http://cipsm.hpc.pub.ro/traffic

Account Information

Name
Fratila Catalin

Username
catalin.fratila@cti.pub.ro

Password

Change Password?
Do you want to change Password?

Update Informations

Update
Send account information to server?

Speed	Max speed
0 kilometers / hour	147 kilometers / hour

Length
817.32
kilometers

Figure 7: Interfaces from the TrafficCollector application.

CAPIM's context collection services), we also developed a Website where users can view aggregated personalized statistics of their own data traces. The mobile application is responsible for collecting the data, and has limited capabilities to present simple statistics (like the maximum travelling speed, the distance covered, average speed and current speed). The data can be further sent and aggregated over longer time intervals on the server-side. In fact, data coming from multiple sources is used to generate a model of the traffic at the city-level, which can be used futher to determine the congestion status for different roads in the city.

We allow users to view their current location based on a unique id. For privacy, the user can also hide its identity.

Examples of graphical user interfaces within the application are presented in Figure 7.

The server supports simple HTTP POST requests. It can hold a large number of uploads without the users seeing any delays, also there are some fail-safe mechanisms: for example, if the server or the connection fail, the Android app still saves the data on the local SQLite and file storage (so when the server is back online the data will still be available).

In terms of energy consumption, the data exchanged between the client and the server is compacted. We studied the impact of the application on the battery and on network traffic (for 3G and real-time mode). We made several experiments, where we evaluated the battery consumption over a period of usage: for example, in 2 hours of using the application when driving the battery decreased with about 10% (we used Samsung Galaxy S smartphones - statistics report that its battery life supports 7 hours of talk time and approx. 312 hours of idle time). At this moment, the real-time mode of the Traffic Collector application sends only data that is not stationary (the speed is greater than zero).

The application needs to be ready for use by many people (Google uses millions of users) to generate large amount of data that can be accessed and analyzed accordingly. Thus, we next present a service designed to support the intelligent storage and processing of such large quantities of sensed data.

6. Case study: intelligent data storing service

In all previous scenarios we had to face one important challenge: how to efficiently store high volumes of data which need to be available in real-time, and support the user's need for events even in high-mobility situations. For this, we propose Context Aware Framework, a service specifically designed to cope with such Big Data challenges. Its architecture is presented in Figure 8 and includes several components (see Figure 8): Data and Metadata Clients, Brokers, the Metadata Manager, and a Cloud-based storage layer.

The *Metadata Client* and *Data Client* connect Context Aware Framework with context-aware applications. Both these software components are part of CAPIM's storage layer. The *Metadata Client* is responsible for creating and accessing the metadata information that describes the context data schema used by a particular application. We acknowledge that different context-aware applications have different requirements in terms of the data scheme used internally. Consequently, this component allows each application to use a different data schema to model the context.

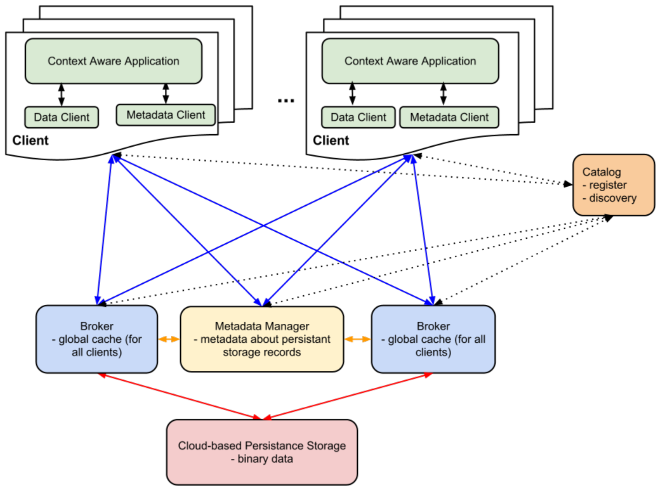

Context Aware Application
Data Client
Metadata Client
Client
...
Context Aware Application
Data Client
Metadata Client
Client
Catalog
- register
- discovery
Broker
- global cache (for all clients)
Metadata Manager
- metadata about persistant storage records
Broker
- global cache (for all clients)
Cloud-based Persistance Storage
- binary data

Figure 8: The proposed architecture.

A *Data Client* can write, retrieve, and store context data necessary to a particular context-aware application. Here we assume a one-to-one relation, each application being served by a dedicated *Data Client*. Each *Data Client* is responsible for supporting the mobility of the user, supporting seamless access to the nearest *Broker*. The *Data Client* works with its own local cache - used for offline situations, when the user cannot connect to the *Broker*.

A dedicated *Discovery Service* is used for the registration and discovery of the existing *Metadata Manager* and *Brokers*. In the architecture we assume the existence of one *Metadata Manager*, but several *Brokers*. The *Discovery Service* is, therefore, also responsable for finding the *Broker* most convenient for a particular *Client*.

The *Metadata Manager* manages the connections between the meta-information describing the data, and the information regarding the actual physical data storage. When a new context-aware application is registered for the first time, the *Metadata Client* connects to the *Metadata Manager* and writes meta-information describing that particular application. The information contains, among others, the datatype formats to describe the context data collected/stored by the application. Next, when the *Data Client* writes context data, it connects to the nearest *Broker*. The context data is sent to the *Broker*, which in turn writes it to the *Persistance layer*. The process involves two steps: first the *Client* writes the data, and next asynchronously the *Broker* handles transparently (in background) the actual writing into the *Persistance layer*. The *Broker* also writes information describing the physical storage parameters to the *Metadata Manager*. For persistent storage we decided to adopt the use of Blobs. A Blob stores the context data needed by one context-aware application. The *Metadata Manager* actually links the meta-information all the way to a particular Blob and to an offset inside it where the particular data resides. The *Metadata Manager* also manages the relation between the persistent Blobs (the ones used for history preservation of context data) and the *Brokers*, where the real-time information is preserved.

The Context Aware Framework assumes the existence of several distributed *Brokers*. The *Broker* is responsible for handling real-time guarantees specified when an application wants to access context data. The *Broker* handles requests coming from a limited number of users, grouped based on their locality. It supports distributed writing of data, and processing of requests coming from clients.

For accessing the context data, a *Client* application generates a filter (expressing the parameters of interest) for finding it. This filter is further received and processed by the *Broker*. The resulting data is sent back to the Client, and is also temporarily stored in a cache, local to the *Broker*. This cache is used to speed up the response time for subsequent requests for similar data. If another client sends a similar request, the *Broker* is capable to reply directly with the data from its own cache (unless the data was invalidated by a subsequent write).

For the *Persistence layer* we use the BlobSeer [23] storage distributed system. BlobSeer consists of a set of distributed entities that cooperate to enable a high throughput storage. Data providers physically store the blocks corresponding to the data updates; new providers may dynamically join and leave the system. The provider manager keeps information about the available storage space and schedules the placement of newly generated blocks, according to a load balancing strategy. Metadata providers store the information that allows identifying the blocks that make up a version of the data. The version manager is in charge of assigning version numbers in such a way that serialization and atomicity are guaranteed. In addition, clients can access Blobs with full concurrency, even if they all access the same Blob. One can get data from the system (*read*), update it by writing a specific range within the Blob (*write*) or add new data to existing Blobs (*append*). Rather than updating the current pages, each such operation generates a new set of pages corresponding to a new version. Metadata is then generated and "weaved" together with the old metadata in such way as to create the illusion of a new incremental snapshot; this actually shares the unmodified pages of the Blob with the older versions.

For Context Aware Framework, Clients can sometimes access the second to the last version of the context-aware data until one write-in-progress operation is finished. Context Aware Framework uses this to provide to the higher-lever applications eventual consistency support for read/write operations.

Typical context-aware applications [4] usually generate big amounts of unstructured or semistructured data. Applications can interpret this data in particular ways, by defining appropriate meta-information associated with it. The applications can decide on their own different granularities - for example, an application can write several chunks of data at once, for the data corresponding to several events, and define one single meta-entry to describe this. It is entirely left in the responsibility of the application to define its use and schema corresponding to the context data and associated model.

6.1. Overall Architectural Benefits

The architecture of Context Aware Framework brings several capabilities:

The framework is designed for context-aware applications that work with data represented mostly as *time-based series*, where entries are in the form $\langle timestamps, Object\rangle$.

The architecture supports *scalable* applications. Once deployed, the system can support a large number of applications, involving potentially large number of users, each with its own context data. This is because each application runs in a separate environment.

Context Aware Framework provides *Locality, Mobility and Real-time access guarantees*. In order to have good response times, a client will connect to the closest *Broker* before launching a request. All read operations are cached on two levels: one is on the *Data Client* side, and one on the *Broker* side. If two clients issue the same request, the response for the second one will be fetched from the Broker's cache. This ensures both a good response time.

Persistence is also supported. Clients write their data, which in turn is saved in the storage system (where we use BlobSeer).

Later on, clients can ask for data, through *complex search filters*. Also, we support *Prediction*. In order to benefit from the large amount of stored data, clients can activate predictions for a specific set of data. When this is hapening, the context data is pre-fetched on the Broker cache, based on complex prediction algorithms. This can be used to cache in advance data for certain data types.

6.2. Implementation

As explained, the *Metadata Manager* is responsible for handling the logical relation between the description of the context data and its actual physical storage. To illustrate this, we assume the following example: in a large city many users might send GPS data to collaboratively support an application capable to aggregate this information and offer a traffic model. Some users are capable to also send data about pollution (they have sensors for monitoring the air quality). We assume this information is sent and stored using the previously described Context Aware Framework.

First, the *Client* will write in the *Metadata Manager* the datatypes used by the application:

```
object Location
    float lat, long;
    string hw_description;


object COLevel
    float level;
    string hw_description;
```

Next, different *Clients* will write the actual context data:

```
arrayTimestamp, Location ==>
 243452343L, 14.5, 34.45, 'Nexus Galaxy',
 243452354L, 14.51, 34.467, 'Nexus Galaxy',
 243452368L, 14.53, 34.473, 'Nexus Galaxy'


arrayTimestamp, COLevel ==>
```

```
 243452344L, 45.3, 'Air Quality Sensor',
 243452360L, 45.4, 'Air Quality Sensor',
 243452412L, 45.37, 'Air Quality Sensor'
```

The data is written in a Blob, inside the *Persistence layer* - the actual data is stored in *BlobSeer*. In this example, the data is written in bursts. We support this feature in cases, for example, when a car can collect data and sent it only when a WiFi connection becomes available. The actual information used to describe the physical storage looks similar to:

```
 UUID, BlobID, BlobVers, BlobOffs, Size
```

, where *UUID* refers to the application id that generated the data, *BlobID*, *BlobVers*, *BlobOffs* and *Size* identify the blob, its version, the data offset and size in the Blob where the information was written. Next, the *Metadata Manager* adds an entry linking the *UUID* to the

```
TimestampStart, TimestampEnd, DataType, UUID,
 BlobID, BlobVers, BlobOffs, Size, NoRecords
```

A concrete realisation of such an entry is:

```
243452343L, 243452368L, 'Location', 0x242,
   213412L, 34, 0, 1234402L, 3
```

The actual implementation of the *Metadata Manager* uses Mongodb [27], a flexible open source document-oriented NoSQL database system. Mongodb includes support for master-slave replication and load balancing. For searching, it also supports regex queries. For Context Aware Framework, the database system was preferred for several reasons: The number of entries kept by the *Metatada Manager* - entries previously described - is small. Each entry follows a structured object-oriented data schema. Consequently, an object-oriented database model is preferred.

Also, when the number of metadata access requests becomes high enough, the system should be able to scale. MongoDB, the distributed object relational database, is the natural choice, because it support distributed deployment and high scalability [27].

The Metadata Manager is also collaboratively used by different applications. For security and management reasons, in the actual implementation each application stores its related data in separate sandboxes.

6.3. Filtering

As previously described, for accessing the data the *Client* builds a search filter. This can include different custom data types defined by an application. The filter specifies the restrictions for searching particular datatypes. For instance, a filter can include restrictions for retrieving specific location and pollution levels. In this example, the filter looks similar to:

```
class Filter
    Location l;
    COLevel c;
    ...
    bool filter()
        return  l.lat > 10.53 and
            l.long < 20.45 and
            c.level < 15;
```

The filter result is in format $(timestamp, location, level)$.

The *Client* sends the serialized version of this filter class, and the *Broker* loads it and instantiate it with values that match the implementation of the filter instance.

6.4. Prediction

Prediction is done using linear interpolation (Lagrange interpolation was used in the pilot implementation). The prediction module is extensible, and the user/application can easily replace it.

For predicting a future value based on a time-dependent series, the user specifies several parameters. The predictability pattern specifies how the data varies (possible values include daily, weekly, or hourly patterns). For example, a daily pattern considers that data is similar for the same hours each day, while a weekly pattern assumes data is similar for the same days each week.

To optimize the prediction process, we consider that only a subset of all data in history is used by the prediction algorithm (a time-window like approach, considering only the last most relevant values). The interpolation considers the set of last values and depends on the type of prediction pattern selected.

To use this facility, the API allows the user to specify N, and two timestamp values. N specifies the number of predicted values the user is requesting - and it is used to define the granularity of the sampling history data. The two timestamps specify the interval in the future of interest for the prediction - the prediction returns in this case the N values spread over the requested interval, by mediating the obtained predicted values. Obviously, if the prediction cannot be performed (or the error is too high), the returned answer can be also none.

We applied the prediction facility to implement an adaptive cache. Such a cache is filled with values that it predicts the user will need in the near future - thus, it can support the losing of the connectivity, or it can support an optimization by requesting data asynchronously from the persistence layer before the actual request for data takes place.

Let us consider the example of an application requesting data about weather. In this case weather is considered to be a function of hour and location. A predictive cache could predict the location of the user in the near future, as well as the time moment he/she will get there. Thus, it will be able to further interrogate a weather service and request the weather values in advance. We tested this assuming that a client requests a new weather value every 30 minutes, and the cache replenishes the weather values in advance, such that by the time client makes the actual request, the cache is able to opportunistically serve him/her the data (i.e., even in case an Internet connection is no longer in place).

6.5. Experiments, Evaluation and Results

For evaluating Context Aware Framework, we used the following scenario: many taxis from a city are equipped with mobile devices that run a context-aware application. This application collects GPS data, and sends it to a server. Clients are presented with context-aware capabilities, such as searching for nearby free taxis or inspecting routes (for example, the municipality can learn the popularity of routes).

As input data, we used a real-world dataset publically available on CRAWDAD [28]. The dataset contains mobility traces of taxi cabs in San Francisco, USA, in the form of GPS coordinates for approximately 500 taxis collected over 30 days in the San Francisco Bay Area (it includes approximately 11 millions unique entries).

These taxis were considered as clients for our context-aware middleware. They were able to write data, and use different access patterns to obtain context-based information. Each client runs on a different node inside a distributed system. For these experiments we used Grid'5000 [29], a large-scale distributed testbed specifically designed for research experiments in parallel, of large-scale distributed computing and networking applications.

To evaluate the performance of Context Aware Framework we had to first filter the data for each unique taxi in the experiment. Therefore, we used 500 different input files, with an average of approximately 20.000 records per file. Each record is specified as [latitude, longitude, occupancy, time]. For example, a record is expressed as [37.75134 -122.39488 0 1213084687], where latitude and longitude are in decimal degrees, occupancy shows if a cab has a fare (1 = occupied, 0 = free) and time is in UNIX epoch format.

For the storage layer, we used BlobSeer. The total data written by each taxi is approximately 5MB.

In Grid'5000 we used 112 dedicated parallel nodes for the clients, and 4 other dedicated parallel nodes for 4 *Brokers*. One other dedicated node was used for the *Metadata Manager*, and another one for *BlobSeer*. In these experiments we used an increasing number of *Brokers* - ending with the 4-based *Broker* experiment. We assume the city is equally split between these *Brokers* - if a taxi always connects to the nearest *Broker*, the mobility data is equilibrated such that we obtained an approximately even number of data sent to each *Broker*. Thus, the number of clients distributed per broker is uniform.

During the experiment we varied configuration parameters such as: the parameters used for BlobSeer configuration (number of data providers, and page size was progressively increased up to 12 MB), the number of clients and brokers, the maximum records written per chunk. We were particularly interested in time taken to perform different operations (to illustrate the capability to support real-time traffic), as well as in the consumed data traffic (to evaluate the optimization obtained when adding the caches).

First, we evaluated the *writing* performance. For this we conducted several experiments where we increased the number of clients that write data (entire input files) to Context Aware Framework. Since we used 112 dedicated nodes, the evaluation is relevant up to this limit - the results are presented in Figure 9. Figures 10a and 10b show the result obtained for different read access patterns. Compared to these figures, the write operation is more time consuming. Still, the time increases by small amounts, thus the system shows good scalability results.

For evaluating the *read* operations, we considered two different scenarios.

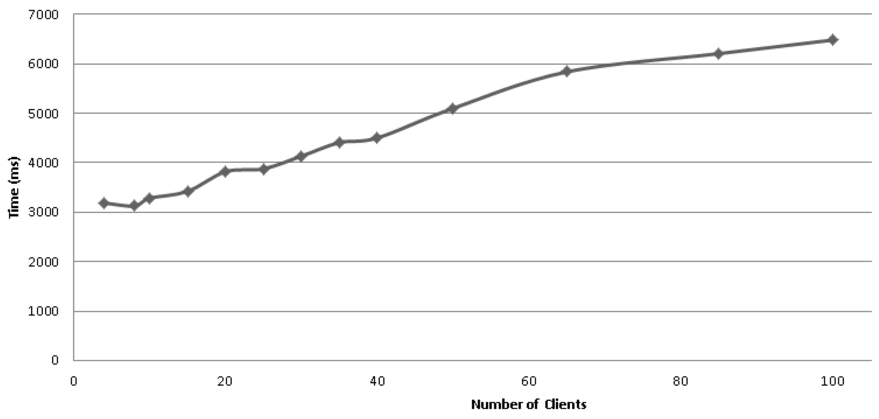


Figure 9: Write Test.

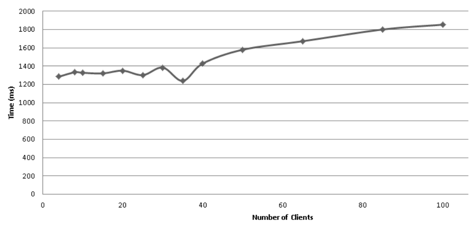
First, a simple search consists in a query where a driver wants to obtain all data relevant for a particular location (given as latitude and longitude limits) and time period. A more complex search operation is one where a client queries the system for the nearest free taxi considering a particular time moment and location. For such a query the system has to aggregate data from two different data types.

Again, we varied the number of clients assumed in the experiment, up to 112. The experiment ran until Context Aware Framework has all the context data persistently written. When all data is written, next all clients issue a filter such that all queries will always return results. In a first experiment, we used the same filter, but the caches will return always the value and the time penalty is minimal. We next assumed that each client issues a unique filter, thus each query is served by questioning the last layer: *BlobSeer*. We were interested to see the Broker's capability to support parallel client requests. Figures 10a and 10b show the results obtained in this case.

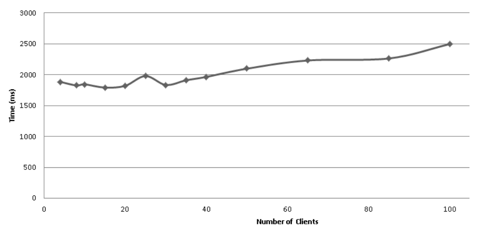
Again, in this case Context Aware Framework is able to successfully handle the queries coming from distinct clients. The results show that time increases by small amounts, thus the system shows again good scalability results.

6.6. Evaluation of prediction

Next, we evaluated the prediction component. In this experiment the prediction is activated for each taxi within the dataset. We splitted the input data file in two parts: 80% of the data was used as input for learning, and



(a) Simple Search.



(b) Complex Search.

Figure 10: Results for searching.

then 20% of the data was used for the evaluation of the prediction accuracy. The predictor in this case uses the data to predict where a cab will be for future time moments, considering daily repeatability patterns - it can be used by a client to search for the nearest taxis, for example, at a future moment of time. In this case, as mentioned, we were particularly interested to measure the prediction accuracy.

The results in Figure 11 show a cumulative graph for number of values passing a prediction acceptance threshold. A threshold of 10% means, for example, that for a variation range of 40km, a value predicted with a 4km error is still accepted as being correct. For the experiment, a 10% threshold means that a value is predicted such that, when compared to the real observed value in the input file, it gives a variation of no more than 10% of the entire city area, assuming that each car drives through the entire city during its experimental lifetime and has an equal probability to be at a certain moment of time in any of the next probable locations.

Looking at Figure 11, when accuracy acceptance percent goes down, there is a random factor that determines some of the values to still be correct; when the percent goes up, there are some "unpredictable" values that make the prediction slightly lower then 100%.

Also, it can be observed that a good threshold is around the value 0.2 for accuracy acceptance, where the prediction becomes very good (yielding approximately 80% correct predicted results).

We next varied the prediction type (considering hourly or weekly patterns). We observed that the correct prediction behavior is similar, but it depends on the nature of the dataset and assumed prediction pattern. For example, predicting with one hour pattern for a too large time interval results in inaccurate prediction results, because the cabs' moving patterns is

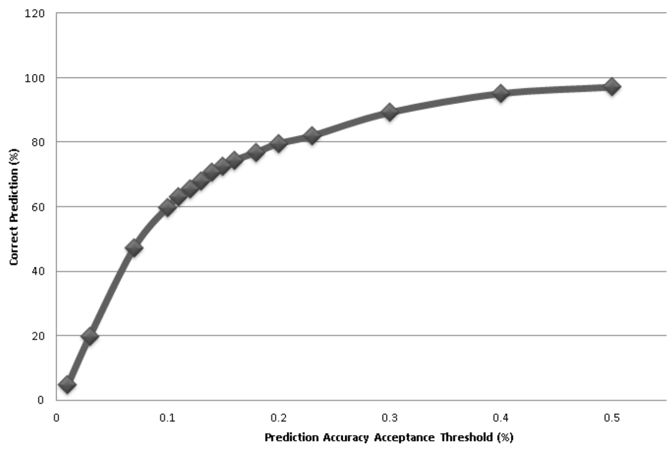


Figure 11: Accuracy Acceptance.

not hourly based (8am traffic, for example, is different than the 11am one).

6.7. Predictive cache

A good use of the prediction module consists in the implementation of a predictive cache that can be used by a mobile application. The cache sits on the mobile device, tightly coupled with the application, and uses prediction to obtain in an opportunistic way data from the storage layer.

We assumed first an experiment designed to test the prediction accuracy of the predictor in a real application. In this experiment, from time to time (e.g., once an hour), a background process asks for a predicted value for the location (e.g., using a pattern such as 'predict my location after an hour'). Then, the process uses this predicted location to ask further for the weather, having both the location and the time for the next interval (hour). Then, the answer is saved into the cache. So, after an hour, the user can ask for the weather in his/her locations, and the answer can be found in the cache with a 80-100% location prediction accuracy.

These experiments were done on a machine with the following characteristics Core i5, 2.5 GHz, 4GB RAM. Unlike the next series of experiments, in this case we assumed that all Clients are always connected to the Internet (and, thus, can access at all times the Broker).

To test the implementation, we have used one Broker. Clients are periodically (every 30 minutes) asking for their predicted location. The obtained results (Figure 12) show that the answer time increases logarithmicaly with

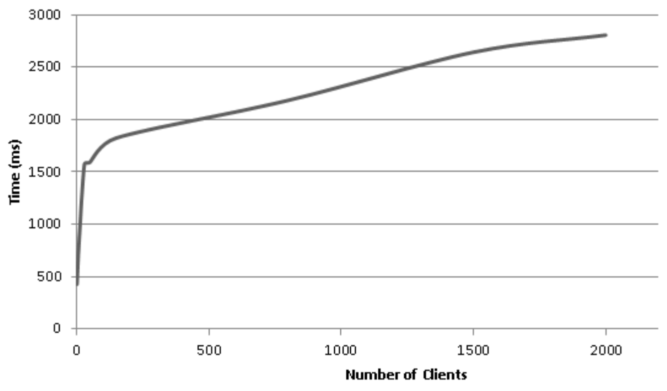


Figure 12: Predictive Cache.

the number of clients; thus, we can say that the predictive cache scales very well for a big number of clients.

For evaluating this capability, we next simulated an application that runs on the user's mobile device and present him/her with traffic information. This kind of data is context-aware because the user is interested to receive traffic information depending on his/her both time and location.

The scenario consists in cabs from San Francisco moving inside the town and trying to acquire the traffic information using a public service. Their only way to connect to the Internet is using WiFi hotspots (3G/4G is too expensive for a large scale system), distributed in a grid configuration through all the town.

The grid was chosen because it provides a good covering of the town with fewer resources than other configurations. The active area of the town is around 250 square km, taking into account our users moving pattern from the input dataset.

We assume that the prediction component is available in the form of a Web service, reachable over the Internet. In our scenario we assumed users ask for new traffic information every 30 minutes (access the Web service).

We also assumed the traffic data does not dependent necessarly on the real-time information. For a typical traffic navigator, the traffic data is generally served by aggregating the traffic data for a certain period of time. This assumption was needed because we assume the information is still valid, even if it is kept in cache for 30 minutes.

Next, we envisioned the following scenario:

From time to time (30 minutes), the user tries to access relevant traffic information (related to his/her time and location). In the implementation this is accomplished by a background process continuoulsy waking up periodically in order to ask the Context Aware Framework about the most “possible“ future location of a particular car. This process, which actually simulates the behavior of the Client cache, then downloads traffic information related to his/her future predicted location - for this, it sends to the traffic predicton Web service the time in future for which it wants the information. The request will be served only if there is an Internet connection available at the moment the request is issued. If not, the service will fail to bring results. If successful, the returning pair (future time, future location) will be locally cached (on the Client cache).

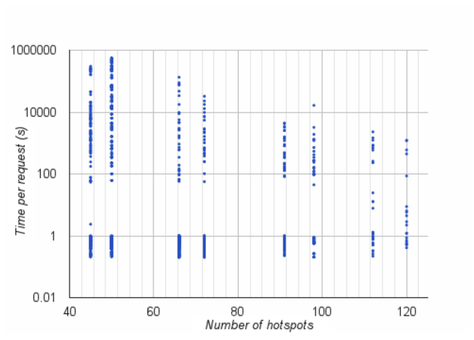
When the client will actually need the traffic data, if the predicted value for its future location was computed correctly it will actually use the cached data. This means that this client will not need an Internet connection to access this new data, and it will have it fast (since it is already cached locally).

Because traffic information is very sensitive to the current user location, we considered in our experiments relevant only location values predicted with an accuracy error lower than 5%.

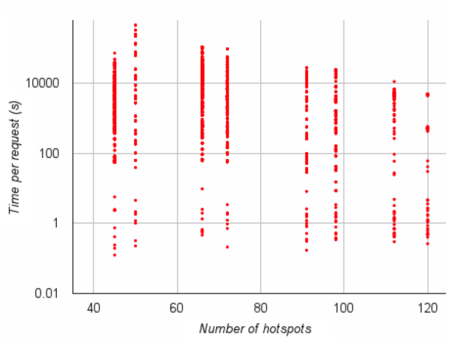
Considering the scenario and the experiment conditions described above, we plotted the average time for one request, having the predictive cache on or off. In the experiment, we varied the number of hotspots in the town's WiFi grid, from 40 to 120 different access points. The obtained results are presented in Figures 13a and 13b.

First, comparing the case when the cache is active and the one when it is not active, we can observe the following. When cache is active the time necessary to serve each request is actually decreasing. When the cache is active, a large amount of requests are finishing under 1 second, with or without Internet connection. When we stop the cache, only requests which are issued by clients within WiFi coverage zones are still served within good time limits, while for the others taxis are not capable to aquire the data until they reach Internet connectivity.

Since there is a compromise between the time for a request and the density of WiFi hotspots, as seen in the plots, the number of hotspots has an important impact over how well the queries are served when using the predictive cache. However, we cannot assume a too much density of such hotspots, considering our scenario that covers only approximately 250 sq km. For in-



(a) Predictive Cache ON.



(b) Predictive Cache OFF.

Figure 13: Predictive Cache Results.

stance, a more detailed analysis for 66 hotspots, when we vary the acceptance level of the prediction to 0.13 (if the predicted location doesn't need to be so precise) revealed that only 12.5% of the requests need an Internet connection (the rest of them were cache hits). This, combined with the fact that only 20% of requests are issued when cabs have Internet connection, leads to only 10% probability for a request not to be solved at the moment it was made.

7. Conclusions and Further Work

Data-intensive computing is now starting to be considered as the basis for a new, forth paradigm for science. Two factors are encouraging this trend. First, vast amounts of data are becoming available in more and more application areas. Second, the infrastructures allowing to persistently store these data for sharing and processing are becoming a reality. This allows unifying knowledge acquired through the previous three paradigms for scientific research (theory, experiments and simulations) with vast amounts of multidisciplinary data. The technical and scientific issues related to this context have been designated as the "Big Data" challenges and have been identified as highly strategic by major research agencies.

In this paper we presented incrementally services designed to support several hot challenges related to Big Data management, by focusing on a particular class of applications: context-aware data-intensive applications. A representative application category is that of Smart Cities, which covers a large spectrum of needs in public safety, water and energy management, smart buildings, government and agency administration, transportation, health, education, etc. Today, many Smart City applications are context-based and event-driven, which means they react to new events and context changes. Such applications have specific data access patterns (frequent, periodic or ad-hoc access, inter-related data access, etc.) and address specific QoS requirements to data storage and processing services (response time, interrogation rate, etc.). With the advent of mobile devices (such as smartphones and tablets) that contain various types of sensors (like GPS, compass, microphone, camera, proximity sensors, etc.), the shape of context-aware (or pervasive) systems changed. Previously, context was only collected from static sensor networks, where each sensor had a well-defined purpose and the format of the data returned was well-known in advance and could not change, regardless of any factors. Nowadays, mobile devices are equipped with multimodal sensing capabilities, and the sensor networks have a much more dynamic behavior due to the high levels of mobility and heterogeneity.

Context Aware Framework, in particular, is designed to support such requirements. In a pervasive world, where the environment is saturated with all kinds of sensors and networking capabilities, support is needed for dynamic discovery of and efficient access to context sources of information. Such requirements are mediated in our case through a dedicated context management layer, which is responsible for discovering and exchanging context information. We presented the context storage system architecture for data management that includes an additional set of components. This supports the mapping between meta-information (describing the context) and the actual context data stored in BlobSeer, data caching and handling requests coming from a distinct set of users or city area, and connecting the metadata management layer to context-aware applications. In addition, we presented a layer that is responsible for creating and accessing the metadata information that describes the context data schema used by a particular application and allows the mobile application to write, retrieve, and store context data. It is also responsible for supporting user's mobility. The components support several requirements: user's mobility and provisioning of data according to his/her locality; real-time guarantees for data provisioning; allow efficient access to the data in terms of speed of access, as well as support for complex queries; discovery and registration of data sources and access to data using different granularities; and scalability.

References

[1] Setis, european initiative on smart cities, http://setis.ec.europa.eu/implementation/technology-roadmap/european-initiative-on-smart-cities, 2013. [Accessed March 9th, 2013].

[2] Navigant research, smart cities, http://navigantresearch.com/research/smart-cities, 2013. [Accessed March 14th, 2013].

[3] W. H. Dutton, K. L. Kraemer, J. G. Blumler, Wired cities: Shaping the future of communications, Macmillan Publishing Co., Inc., 1987.

[4] C. Dobre, Capim: A platform for context-aware computing, in: Proceedings of the 2011 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 266–272.

[5] A. K. Dey, G. D. Abowd, D. Salber, A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications, Human–Computer Interaction 16 (2001) 97–166.

[6] P. Fahy, S. Clarke, Cass–a middleware for mobile context-aware applications, in: Workshop on Context Awareness, MobiSys, Citeseer.

[7] J. E. Bardram, The java context awareness framework (jcaf)–a service infrastructure and programming framework for context-aware applications, in: Pervasive Computing, Springer, 2005, pp. 98–115.

[8] H. Chen, T. Finin, A. Joshi, L. Kagal, F. Perich, D. Chakraborty, Intelligent agents meet the semantic web in smart spaces, Internet Computing, IEEE 8 (2004) 69–79.

[9] T. Gu, H. K. Pung, D. Q. Zhang, A service-oriented middleware for building context-aware services, Journal of Network and computer applications 28 (2005) 1–18.

[10] K. Boloor, R. Chirkova, Y. Viniotis, T. Salo, Dynamic request allocation and scheduling for context aware applications subject to a percentile response time sla in a distributed cloud, in: Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on, IEEE, pp. 464–472.

[11] A. M. Aly, A. Sallam, B. M. Gnanasekaran, L.-V. Nguyen-Dinh, W. G. Aref, M. Ouzzani, A. Ghafoor, M3: Stream processing on main-memory mapreduce, in: Proceedings of the 2012 IEEE 28th International Conference on Data Engineering, ICDE '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 1253–1256.

[12] T. M. Ghanem, A. K. Elmagarmid, P.-A. Larson, W. G. Aref, Supporting views in data stream management systems, ACM Trans. Database Syst. 35 (2008) 1:1–1:47.

[13] S. L. Kiani, A. Anjum, M. Knappmeyer, N. Bessis, N. Antonopoulos, Federated broker system for pervasive context provisioning, 2012.

[14] Google now, http://www.google.com/landing/now/, 2013. [Accessed February 9th, 2013].

[15] Microsoft onX, http://www.onx.ms, 2013. [Accessed March 9th, 2013].

[16] G. Chen, D. Kotz, et al., A survey of context-aware mobile computing research, Technical Report, Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, 2000.

[17] K. Henricksen, A framework for context-aware pervasive computing applications, University of Queensland, 2003.

[18] M. tim jones, anatomy of a cloud storage infrastructure. models, features, and internals, www.ibm.com/developerworks/cloud/library/cl-cloudstorage/, 2010. [Accessed March 12th, 2013].

[19] B. McBride, Jena: a semantic web toolkit, Internet Computing, IEEE 6 (2002) 55 – 59.

[20] Foaf website, http://xmlns.com/foaf/spec, 2013. [Accessed March 9th, 2013].

[21] Geo website, http://www.geonames.org/ontology/, 2013. [Accessed March 9th, 2013].

[22] Wail website, http://www.eyrie.org/~zednenem/2002/wail/, 2013. [Accessed March 9th, 2013].

[23] B. Nicolae, G. Antoniu, L. Bougé, Blobseer: how to enable efficient versioning for large object storage under heavy access concurrency, in: Proceedings of the 2009 EDBT/ICDT Workshops, EDBT/ICDT '09, ACM, New York, NY, USA, 2009, pp. 18–25.

[24] N. Bessis, F. Xhafa, Next Generation Data Technologies for Collective Computational Intelligence, volume 352, Springer, 2011.

[25] N. Kaufmann, T. Schulze, D. Veit, More than fun and money. worker motivation in crowdsourcing–a study on mechanical turk, in: Proceedings of the Seventeenth Americas Conference on Information Systems, Detroit, MI.

[26] Traffic collector, http://play.google.com/store/apps/details?id=ro.pub.acs.traffic.collector, 2013. [Accessed March 9th, 2013].

[27] R. Hecht, S. Jablonski, Nosql evaluation: A use case oriented survey, in: Proceedings of the 2011 International Conference on Cloud and Service Computing, CSC '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 336–341.

[28] San francisco taxi dataset, http://crawdad.cs.dartmouth.edu/meta.php?name=epfl/mobility, 2013. [Accessed March 14th, 2013].

[29] Grid'5000, https://www.grid5000.fr/, 2013. [Accessed March 9th, 2013].