Paper

Adaptive Distributed Data Storage for Context-Aware Applications

Elena Burceanu, Ciprian Dobre, and Valentin Cristea

Faculty of Automatic Control and Computers, University Politehnica of Bucharest, Romania

Abstract—Context-aware computing is a paradigm that relies on the active use of information coming from a variety of sources, ranging from smartphones to sensors. The paradigm usually leads to storing large volumes of data that need to be processed to derive higher-level context information. The paper presents a cloud-based storage layer for managing sensitive context data. To handle the storage and aggregation of context data for context-aware applications, Clouds are perfect candidates. But a Cloud platform for context-aware computing needs to cope with several requirements: high concurrent access (all data needs to be available to potentially a large number of users), mobility support (such platform should actively use the caches on mobile devices whenever possible, but also cope with storage size limitations), real-time access guarantees – local caches should be located closer to the end-user whenever possible, and persistency (for traceability, a history of the context data should remain available). BlobSeer, a framework for Cloud data storage, is a perfect candidate for storing context data for large-scale applications. It offers capabilities such as persistency, concurrency and support for flexible storage schema requirement. On top of BlobSeer, Context Aware Framework is designed as an extension that offers context-aware data management to higher-level applications, and enables scalable high-throughput under high-concurrency. On a logical level, the most important capabilities offered by Context Aware Framework are transparency, support for mobility, real-time guarantees and support for access based on meta-information. On the physical layer, the most important capability is persistent Cloud storage.

Keywords—Blobseer, brokers, context, distributed, mobile devices communication.

1. Introduction

Today smartphones are becoming commodity hardware. They are seen everywhere, as more people realize that having more sensing and computing capabilities in every-day situations is attractive for many reasons. Smartphones are in fact already used to optimize (e.g., by helping organizing tasks, contacts, etc.) and assist (e.g., with navigation, find information more quickly, access online data, etc.) users with their everyday activities. Their success is the basis for a shift towards developing mobile applications that are capable to recognize and pro-actively react to user's own environment. Such context-aware mobile applications can help people better interact between themselves and with their surrounding environments. This is the basis for a paradigm where the context is actively used by applications designed to take smarter and automated decisions: mute the phone when the user is in a meeting, show relevant information for the user's current location, assist the user find its way around a city, or automatically recommend events based on the user's (possibly learned) profile and interests.

This vision is supported today by the inclusion of context as an active operational parameter of service provisioning. For many mobile applications an important requirement is represented by the active sensing of the operational environment, as users expect to receive only relevant content back on their mobile devices (e.g., when the user accesses a transportation service he expects to receive a route relevant for current location). The advances in mobile technologies support the inclusion of context as an active operational parameter for such applications, as today's mobile devices allow the users to acquire and manipulate complex, multifaceted information in real time, and to interact with each other in seamless ways. The range of applications using context is increasing rapidly, and it spans from urban navigation, cultural heritage to entertainment and peer-to-peer communication. The challenge for such pervasive applications is how to make them continuously adapt to dynamic changes in the environment, even when people move and when the underlying network architecture can offer only limited services.

To support this, the authors designed the Context Aware Framework as a middleware to automate the provisioning of context data to mobile applications. It sits between a persistence layer, where data is actually stored in a Cloud storage system, and the actual context-aware application running on the user's mobile devices, masking the complexity of managing the data. 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 contribution of this paper is twofold: we first introduce the Context Aware Framework middleware, together with the design requirements that motivated our choices; we then present experimental results, supporting our decisions and illustrating the performance obtained when using Context Aware Framework.

The rest of the paper is organized as follows. Section 2 presents an overview of the main relevant work in the field. Section 3 makes an analysis of the main requirements for context-aware storage systems and presents the architecture of the Context Aware Framework that we devised following this analysis. Section 4 presents details of the implementation, while Section 5 presents evaluation scenarios and results illustrating the overall obtained system performance. The paper is concluded in Section 6.

2. Related Work

The ubiquity of mobile devices and sensor pervasiveness call for scalable computing platforms to store and process the vast amounts of the generated streamed data. 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 year 2011 90.9% of data center sites used more storage space than they did three years ago. During that same three-year period, 37% were able to reduce their staff, and 29% 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, today's 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.

Several cross-device context-aware application middleware systems have been developed previously. In their majority these were Web service-based context-aware systems, especially the most recent ones. However, there has been a big variety of middleware systems, developed mainly in the early 2000, that do not rely on Web service technologies and are not designed to work on Web service-based environments [1]. In this work the authors began by studying several popular context-aware platforms, considering their provided functions and particular characteristics. From the category of non-based on web service context-aware platforms the following could be mentioned.

RCSM [2] is a middleware supporting context sensitive applications based on an object model: context-sensitive applications are modeled as objects. RCSM supports situation awareness by providing a special language for specifying situation awareness requirements. Based on these requirements, application-specific object containers for runtime situation analysis will be generated. RCSM runtime system obtains context data from different sources and provides the data to object containers which conduct the situation analysis.

The **JCAF** (Java Context Awareness Framework) [3] supports both the infrastructure and the programming framework for developing context-aware applications in Java. Contextual information is handled by separate services to which clients can publish and from which they can retrieve contextual. The communication is based on Java RMI (Remote Method Invocation). An example of application that use Java RMI is MultiCaR: Remote Invocation for large scale, Context-Aware Applications [4]. This application also address the issue of big data analytics.

The **PACE** middleware [5] provides context and preference management together with a programming toolkit and tools for assisting context-aware applications to store, access, and utilize contextual information managed by the middleware. PACE supports context-aware applications to make decisions based on user preferences.

CAMUS is an infrastructure for context-aware network-based intelligent robots [6]. It supports various types of context information, such as user, place and environment, and context reasoning. However, this system is not based on Web services and it works in a close environment.

SOCAM is a middleware for building context-aware services [7]. It supports context modeling and reasoning based on OWL. However, its implementation is based on RMI.

Web service-based context-aware platforms include the following.

CoWSAMI is a middleware supporting context-awareness in pervasive environments [8]. The **ESCAPE** framework [1] is a Web services-based context management system for teamwork and disaster management. ESCAPE services are designed for a front-end of mobile devices and the back-end of high end systems. The front-end part includes components support for context sensing and sharing that are based on Web services and are executed in an ad hoc network of mobile devices. The back-end includes a Web service for storing and sharing context information among different front-ends.

The **inContext** project [1] provides various techniques for supporting context-awareness in emerging team collaboration. It is designed for Web services-based collaborative working environments. inContext provides techniques for modeling, storing, reasoning, and exchanging context information among Web services.

Being context-aware allows software not only to be able to deal with changes in the environment the software operates in, but also being able to improve the response to the use of the software. That means context-awareness techniques aim at supporting both functional and non-functional software requirements. Authors of [9] identified three important context-awareness behaviors:

– the representation of available information and services to an end user,

– the automatic execution of a service,

– the tagging and storing of context information for later retrieval.

For massive context data streaming applications, M3 [10] 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 [11]. 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 Context Aware Framework, the author of [12] present 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. The main difference compared to our system is that our architecture implies the existence of a metadata manager for all stored information, which relieves much of the burden impose for managing data on the mobile application, and that we 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. The authors 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, our platform can also be compared with Google Now [13] and Microsoft On{X} [14]. Google Now [13] 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. This 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}. Unlike previous work, the platform presented in this paper is specifically designed for the management of large-scale pervasive environments. The platform is designed to support scenarios with potentially thousands of sensors working together for the fine-grain analysis of phenomena. For example, the platform can cope with Smart City context-aware applications and services, providing citizens with real-time information, regardless of their mobility constraints, about current traffic values or other events of interests.

3. Architecture

3.1. Analysis of Requirements for Context-aware Applications

In a typical context-aware application users rely on their mobile devices to receive information depending on their current environment. Such applications can help users augment their reality: they could receive information about neighboring places or buildings in a typical tourism application; they could receive recommendations or navigation data that could help a driver to more easily navigate around a city. In such scenarios users are generally *moving*, and typical context data includes elements such as locality, time, user's status, etc. *Proximity* is 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. Therefore, Context Aware Framework should be able to support user's mobility and provisioning of data according to his locality.

Except for mobility, context-aware applications should provide real-time guarantees for data provisioning. The user should not receive events that happened too far in the past, as such events might not even be valid for his current interests. For example, if a tourist is looking for information about objectives near him, he might not be so happy receiving recommendations for a trip taken some while back. The same might happen when users expect alerts about potential congestions in traffic: if he receives such alerts when is already in the congested road, the information is not so valuable anymore. To cope with such a requirement, Context Aware Framework should support fast dissemination of data between users.

We also acknowledge the imperfections of today's wireless communication infrastructures. Context Aware Framework will not assume the user is always connected to the Internet (e.g., in situations where a wireless connection is not available). It will support the use of context data even when an Internet connection is not available, and in this case we look at alternatives such as opportunistically using the data accessed by others from distributed caches using only short-range communication (in form of Bluetooth and/or Wi-Fi).

The system 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 will 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".

For context-aware applications we consider that the system should support 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.

The framework needs to support scalability. For a typical collaborative traffic application, the number of users could potentially be in the range of millions. The data should be persistently stored. The history of data should be preserved for traceability and advanced data mining processing.

Except for these theoretical requirements, the functional demands coming from a context-aware application was also analyzed. For this we analyzed CAPIM [15], a platform designed to support context-aware services. Such services are designed to help people in an university, who may be endowed with a portable device, on top of which they run an application which facilitates the access to information by automatically reacting to changes of context. CAPIM brings support by:

– providing different information contents based on the different interests/profiles of the visitor (student or professor, having scientific interests in automatic systems or computer science, etc.), and on the room he is currently in;

– learning, from the previous choices formed by the visitor, what information he is going to be interested in next;

– providing the visitor with appropriate services – to see the user's university records only if appropriate credentials are provided, to use the university's intranet if the user is enrolled as staff;

– deriving location information from sensors which monitor the user environment;

– provide active features within the various areas of the university, which alert people with hints and stimuli on what is going on in each particular ambient.

In addition to the previously identified requirements, this scenario validated several new ones in terms of data accesses: 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.

To cope with these requirements, Context Aware Framework should include several layers. First of all, for persistence, collaborative and mobility support, the data should be stored remotely to any mobile device. 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.

BlobSeer [16] 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 it 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 [16] offers an appropriate alternative, as it provides real-time guarantees, large concurrent access guarantees, and support for eventual consistency through an advanced versioning mechanism. This is what motivated our choices in what follows.

3.2. Architecture

Based on the identified requirements, we propose the architecture presented in Fig. 1 is proposed.

The architecture includes several components (see Fig. 2): Data and Metadata Clients, Brokers, the Metadata Manager, and a Cloud-based storage layer.

The **Metadata Client** and **Data Client** connect the Context Aware Framework with the third-party context-aware applications. In the practical implementation (detailed in the next Section) both these software components are integrated into the context-aware application in need of their support (they offer an API for such applications).

The **Metadata Client** is responsible for creating and accessing the metadata information that describes the context data schema used by a particular application. In fact, we acknowledge that various context-aware applications have different requirements in terms of the data scheme used internally. Consequently, each application can use a different data schema to model the context.

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 access anymore the data from its Broker.

A dedicated **Discovery Service** is also 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 **Persistence layer**. The process involves two steps: first the Client writes the data, and next asynchronously the Broker handles

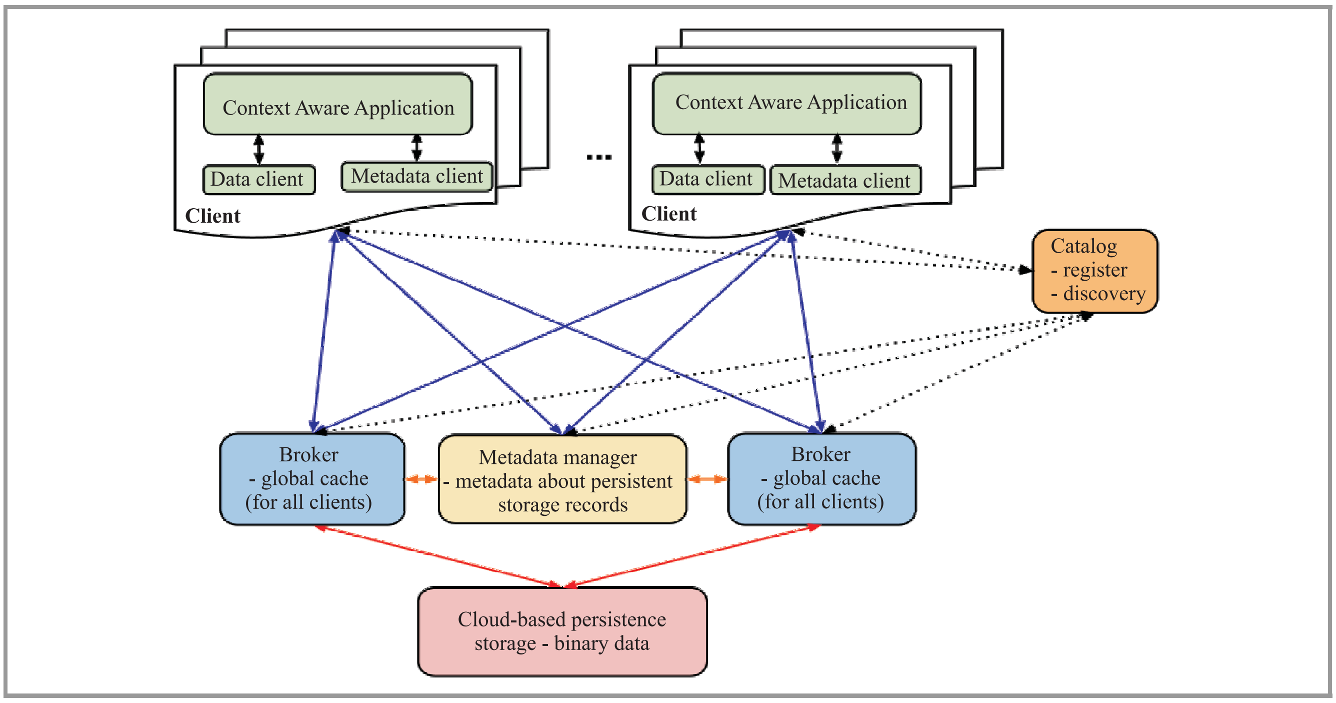

Fig. 1. The proposed architecture.

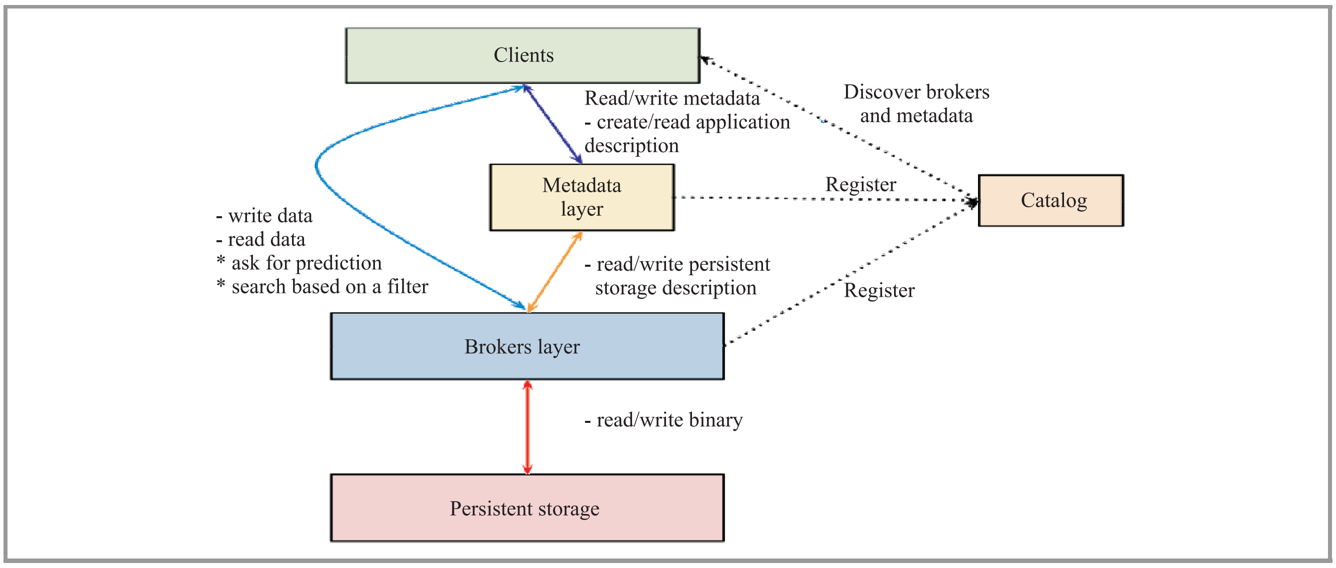

Fig. 2. Architecture layers.

transparently (in background) the actual writing into the Persistence 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 comprises 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 [16] 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 the 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 our 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 [15] 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.

3.3. Overall Architectural Benefits

The architecture of Context Aware Framework brings several capabilities mentioned below.

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, prediction is supported. In order to benefit from the large amount of stored data, clients can activate predictions for a specific set of data (see Section 4.2). When this is happening, 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.

4. Implementation

Next a pilot implementation of the Context Aware Framework architecture previously described was developed.

As explained, the Metadata Manager is responsible for handling the logical relation between the description of the context data and its actual physical storage. In 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, which is similar to:

```
array{Timestamp, Location} ==> {
 {243452343L, {14.5, 34.45, 'Nexus Galaxy'}},
 {243452354L, {14.51, 34.467, 'Nexus Galaxy'}},
 {243452368L, {14.53, 34.473, 'Nexus Galaxy'}}
}

array{Timestamp, 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 Wi-Fi 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}:
```

(e.g.,

```
{243452343L, 243452368L, 'Location', 0x242,
   213412L, 34, 0, 1234402L, 3}).
```

The actual implementation of the Metadata Manager uses Mongodb [17], 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 [18].
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.

4.1. 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.

4.2. Prediction

Prediction is done using linear interpolation (in the pilot implementation Lagrange interpolation was used). The prediction module is extensible, and the user/application can easily replace it.
For predicting a future value based on a time-dependentł;inebreak 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, 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 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's 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 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 the data (i.e., even in case an Internet connection is no longer in place).

5. Experiments, Evaluation and Results

5.1. Experimental Setup

For evaluating Context Aware Framework, the following scenario is used: 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, a real-world dataset publically available on CRAWDAD is used [19]. 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 [20], 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, 500 different input files was used, 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 5 MB.

In Grid'5000 112 dedicated parallel nodes for the clients was used, 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 configuration data was varied, 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.

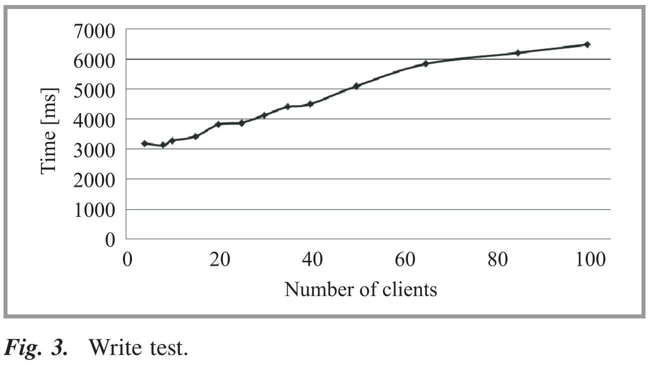

Fig. 3. Write test.

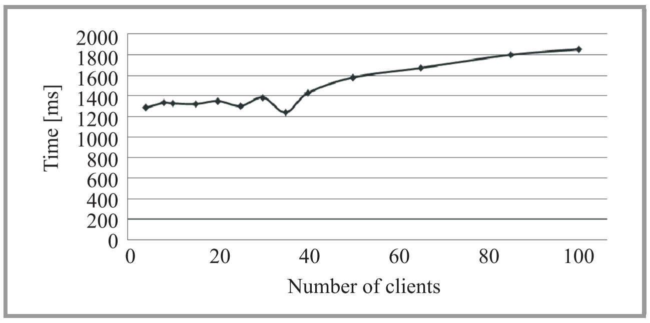

Fig. 4. Simple search.

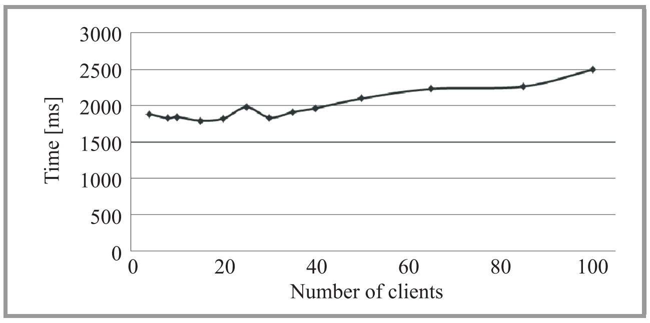

Fig. 5. Complex search.

First, the writing performance was evaluated. For this the authors conducted several experiments where the number of clients that write data (entire input files) to Context Aware Framework was increased. Since 112 dedicated nodes is used, the evaluation is relevant up to this limit – the results are presented in Fig. 3. Figures 4 and 5 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. 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 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 4 and 5 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.

5.2. Evaluation of Prediction

Next, the prediction component is evaluated. In this experiment the prediction is activated for each taxi within the dataset. The input data file is splitted in two parts: 80% of the data was used as input for learning, and 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, the authors were particularly interested to measure the prediction accuracy.
The results in Fig. 6 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 40 km, a value predicted with a 4 km 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.

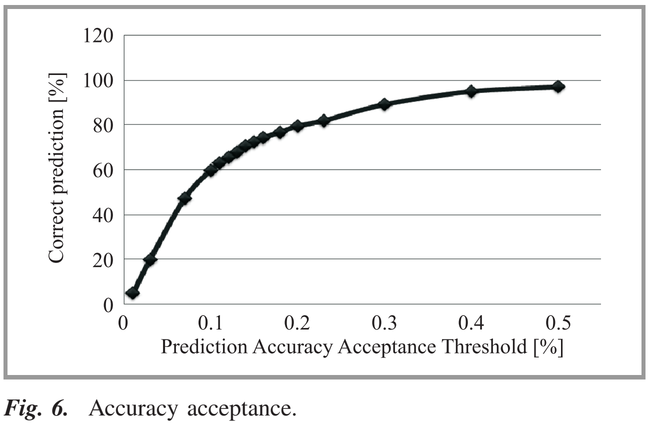

Fig. 6. Accuracy acceptance.

Looking at Fig. 6, 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.
Next the prediction type was varied (considering hourly or weekly patterns). It can be 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 not hourly based (8 am traffic, for example, is different than the 11 am one).

5.3. 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.
First an experiment was 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 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: Intel Core i5 processor, 2.5 GHz, 4 GB 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 (Fig. 7) show that the answer time increases logarithmicaly with the number of clients. Thus, that the predictive cache scales very well for a big number of clients.

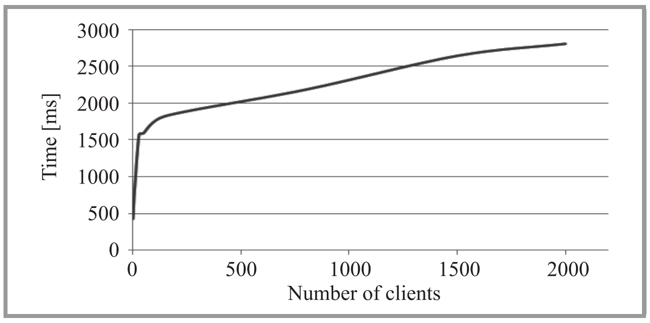

Fig. 7. Predictive cache.

For evaluating this capability, next an application that runs on the user's mobile device and present him with traffic information is simulated. This kind of data is context-aware because the user is interested to receive traffic information depending on his 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 Wi-Fi hotspots (3G/4G is too expensive for a large scale system), distributed in a grid configuration through all the town (see Fig. 8).

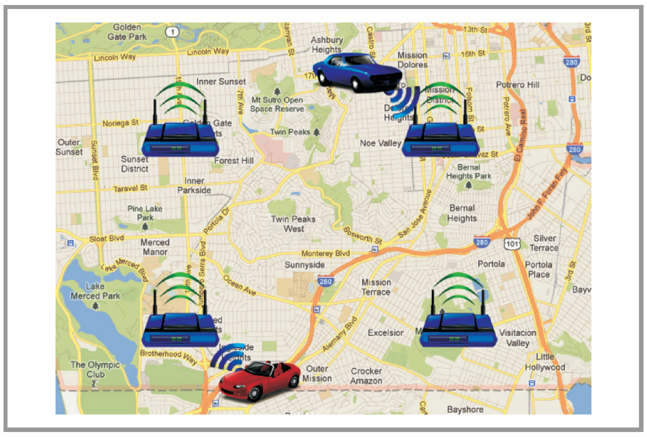

Fig. 8. Taxis connect to the nearest Wi-Fi access points.

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 necessarily 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, the following scenario is envisioned.

From time to time (30 minutes), the user tries to access relevant traffic information, related to his time and location. In the implementation this is accomplished by a background process continuously 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 future predicted location – for this, it sends to the traffic prediction 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, in experiments relevant only location values predicted with an accuracy error lower than 5% was considered.

Considering the scenario and the experiment conditions described above, the average time for one request is plotted, having the predictive cache on or off. In the experiment, the number of hotspots in the town's Wi-Fi grid is varied, from 40 to 120 different access points. The obtained results are presented in Figs. 9 and 10.

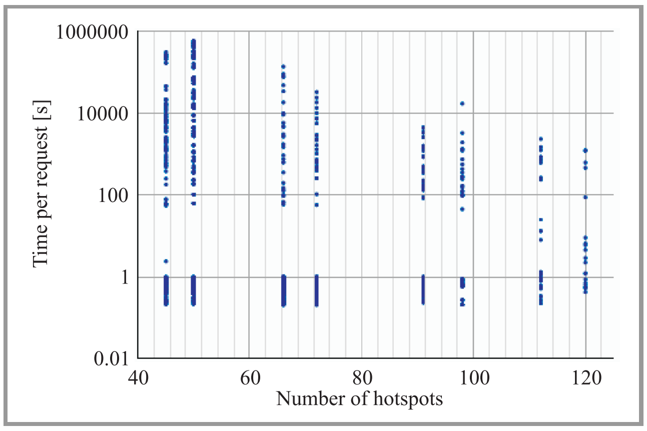

Fig. 9. Predictive cache on.

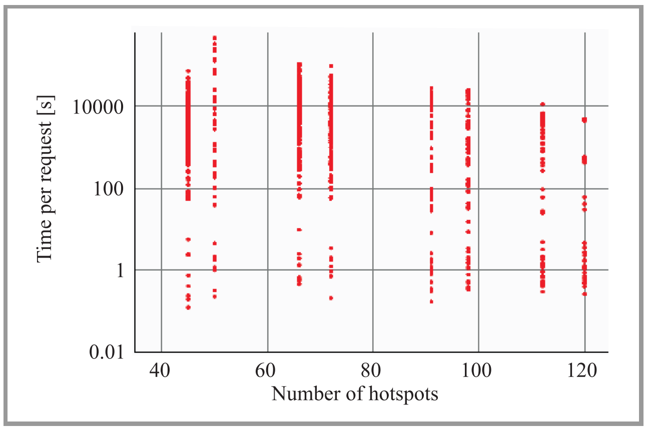

Fig. 10. Predictive cache off.

When cache is active the time necessary to serve each request is actually decreasing. A large amount of requests are finishing under 1 second, with or without Internet connection. When the cache is stopped, only requests which are issued by clients within Wi-Fi coverage zones are still served within good time limits, while for the others taxis are not capable to acquire the data until they reach Internet connectivity.

Since there is a compromise between the time for a request and the density of Wi-Fi 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 instance, 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.

6. Conclusions and Further Work

In this paper we presented a platform designed to support several hot challenges related to big data management on clouds 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 (i.e., response time, interrogation rate). With the advent of mobile devices (such as smartphones and tablets) that contain various types of sensors (like GPS, compass, microphone, camera, proximity sensors), 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 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. The authors 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 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.

Acknowledgements

The research presented in this paper was supported by the INRIA Associated Team DataCloud@work. This work was also partially supported by project "ERRIC – Empowering Romanian Research on Intelligent Information Technologies/FP7-REGPOT-2010-1", ID: 264207, and by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Romanian Ministry of Labour, Family and Social Protection through the Financial Agreement POSDRU/89/1.5/S/62557. The authors would like to thank Luc Bouge and the entire KerData team. The experiments presented in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed, an initiative of the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS, RENATER and other contributing partners (see http://www.grid5000.fr/).

References

[1] H.-L. Truong and S. Dustdar, "A survey on context-aware web service systems", *Int. J. Web Inform. Syst.*, vol. 5. no. 1, pp. 5–31, 2009.

[2] S. S. Yau and F. Karim, "A context-sensitive middleware for dynamic integration of mobile devices with network infrastructures", *J. Parall. & Distrib. Comput.*, vol. 64, no. 2, pp. 301–317, 2004.

[3] J. E. Bardram, "The java context awareness framework (JCAF) – a service infrastructure and programming framework for context-aware applications", in *Proc. 3rd Int. Conf. on Pervasive Computing PERVASIVE 2005*, Munich, Germany, 2005, pp. 98–115.

[4] G. Cugola and M. Migliavacca, "Multicar: Remote invocation for large scale, context-aware applications", in *Proc. IEEE Symp. Comp. Commun. ISCC 2010*, Riccione, Italy , 2010, pp. 570–576.

[5] K. Henricksen and R. Robinson, "A survey of middleware for sensor networks: State-of-the-art and future directions", in *Proc. 7th Int. Worksh. Middlew. for Sen. Netw. Middleware '06*, Melbourne, Australia 2006, pp. 60–65.

[6] H. Kim, Y.-J. Cho, and S.-R. Oh, "Camus: A middleware supporting context-aware services for network-based robots", in *IEEE Worksh. Adv. Robot. and its Soc. Impacts, 2005, on*, Nagoya, Japan, 2005, pp. 237–242.

[7] T. Gu, H. K. Pung, and D. Q. Zhang, A service-oriented middleware for building context-aware services. *J. Network & Comp. Appl.*, vol. 28, no. 1, pp. 1–18, 2005.

[8] D. Athanasopoulos *et al.*, "CoWSAMI: Interface-aware context gathering in ambient intelligence environments", *Pervasive & Mob. Comput.*, vol. 4, no. 3, pp. 360–389, 2008.

[9] A. K. Dey, G. D. Abowd, and D. Salber, "A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications", *Human-Comp. Interact.*, vol. 16, no. 2, pp. 97–166, 2001.

[10] A. M. Aly *et al.*, M3: Stream processing on main-memory mapreduce", in *Proc. IEEE 28th Int. Conf. Data Engin. ICDE'12*, Washington, DC, USA, 2012, pp. 1253–1256.

[11] T. M. Ghanem, A. K. Elmagarmid, P.-A. Larson, and W. G. Aref, “Supporting views in data stream management systems”, *ACM Trans. Database Syst.*, vol. 35, no. 1, pp. 1:1–1:47, 2008.

[12] S. L. Kiani *et al.*, “Federated broker system for pervasive context provisioning”, *J. Syst. & Softw.*, vol. 86, no. 4, pp. 1107–1123, 2013.

[13] Google now [Online]. Available: www.google.com/landing/now (accessed July 9th, 2013).

[14] Microsoft onX [Online]. Available: https://www.onx.ms/ (accessed July 9th, 2013).

[15] C. Dobre, “Capim: A platform for context-aware computing”, in *Proc. 6th Int. Conf. on P2P, Paral., Grid, Cloud Internet Comput. 3PGCIC 2011*, Barcelona, Spain, 2011, pp. 266–272.

[16] B. Nicolae, G. Antoniu, and L. Bougé, “Blobseer: how to enable efficient versioning for large object storage under heavy access concurrency”, in *Proc. EDBT/ICDT Joint Conf. (12th Int. Conf. Ext. Datab. Technol. & 12th Int. Conf. Datab. Theory) EDBT/ICDT 2009*, Saint-Petersburg, Russia, 2009, pp. 18–25.

[17] R. Hecht and S. Jablonski, “Nosql evaluation: A use case oriented survey”, in *Proc. Int. Conf. Cloud & Service Comput. CSC 2011*, Hong Kong, China, 2011, pp. 336–341.

[18] P. Pääkkönen and D. Pakkala, “Report on scalability of database technologies for entertainment services”, 2012 [Online]. Available: http://virtual.vtt.fi/virtual/nextmedia/Deliverables-2011/D1.2.3.3_MUMUMESE_Report%20on%20Scalability%20of%20database%20technologies%20for%20entertainment%20services.pdf

[19] San Francisco taxi dataset [Online]. Available: http://crawdad.cs.dartmouth.edu/meta.php?name=epfl/mobility

[20] Grid’5000 [Online]. Available: https://www.grid5000.fr/ (accessed July 9th, 2013).

Elena Burceanu received the M.Sc. degree from the University Politehnica of Bucharest. Her main research areas are Intelligent Recommendation Systems, Cloud Data Storage and Large Scale Distributed Systems. The topic of her diploma thesis was in the field of Crowd-based Recommendation, with a particular focus towards Pervasive Systems. This was later continued during her master studies. During that period, as member of the DataCloud@Work Associated Team, she was visiting student with INRIA Rennes Ker-Data team. Her Master research topic addressed the use of context data, gathered using monitoring instruments for Clouds, for opportunistic storing, load balancing and fault tolerance in case of large datasets. She proposed using monitoring data related to the current context, together with augmenting the data with additional semantic information (meta-data), to optimize data location based on different performance metrics. Such metrics can use the semantics, system’s load, network throughput, history of failures, proximity to opportunistic store the large blobs of data to deliver near-optimum response times.

E-mail: elena.burceanu@cti.pub.ro
University Politehnica of Bucharest
313, Splaiul Independentei
Office EG403, sector 6
060042 Bucharest, Romania

Ciprian Dobre has scientific and scholarly contributions in the field of large scale distributed systems concerning monitoring (MonALISA), data services (PRO, Data-Cloud@Work), high-speed networking (VINCI, FDT), large scale application development (EGEE III, SEE-GRID-SCI), evaluation using modeling and simulation (MONARC 2, VNSim). He was awarded a Ph.D. scholarship from California Institute of Technology (Caltech, USA), and another one from Oracle. His results received two CENIC Awards, and three Best Paper Awards, and were published in 6 books, 10 articles in major international peer-reviewed journal, and over 60 articles in well-established international conferences and workshops. He is local project coordinator for national projects: CAPIM – Context-Aware Platform using Integrated Mobile Services, and TRANSYS – Models and Techniques for Traffic Optimizing in Urban Environments.

E-mail: ciprian.dobre@cs.pub.ro
University Politehnica of Bucharest
313, Splaiul Independentei
Office EG403, sector 6
060042 Bucharest, Romania

Valentin Cristea is Professor of the Computer Science Department of UPB. His main fields of expertise are Large Scale Distributed Systems, e-Services, Distributed Systems, Grid Computing. He is the director of the NCIT. He has a long experience in the development, management and coordination of international and national research projects. He participated to the specification of RoDiCA – Romanian Distributed Collaborative Architectures, leaded the PUB team in COOPER and conducted the CoLaborator project for building a collaborative environment for High Performance Computing in Romania. He co-supervised the PUB Team in SEE-GRID-SCI (FP7) and EGEE III (FP7). He is Partner Coordinator in DataCloud@work and in 2003 and 2011 received the IBM faculty award. Prof. Cristea published more than 120 specialist papers, 25 books, and 60 technical reports. He is Ph.D. supervisor, coordinator of the Master program on Advanced Software Services, and the Romanian coordinator of the Master program on Parallel and Distributed Computer Systems co-developed with VU Amsterdam.

E-mail: valentin.cristea@cs.pub.ro
University Politehnica of Bucharest
313, Splaiul Independentei
Office EG403, sector 6
060042 Bucharest, Romania