Distributed Data Storage in Support for Context-Aware Applications

Elena Burceanu*, Ciprian Dobre*, Valentin Cristea*, Alexandru Costan†, Gabriel Antoniu†
*University Politehnica of Bucharest
Bucharest, Romania
E-mails: elena.burceanu@cti.pub.ro, ciprian.dobre@cs.pub.ro, valentin.cristea@cs.pub.ro
†INRIA Rennes-Bretagne Atlantique, IRISA
Rennes-Bretagne, France
E-mails: alexandru.costan@inria.fr, gabriel.antoniu@inria.fr

***Abstract*—**

Context-aware computing is a new paradigm that relies on large amounts of data collected from a variety of sources, ranging from smartphones to sensors, to automatically take smart decisions. This usually leads to large volumes of data, that need to be further processed to derive higher-level context information. Clouds have recently emerged as interesting candidates to support the storage and aggregation of such data for large-scale context-aware applications. However, specific extensions to support context-aware data need to be designed in order to be able to fully exploit the clouds' potential. In this paper we introduce such a cloud-based system, designed to support real-time processing and persistent storage of context data. Context Aware Framework is designed as an extension of the BlobSeer storage system, building a context-aware layer on top of it to enable scalable high-throughput under high-concurrency for big context data. Our experimental evaluation validates the transparency, mobility and real-time guarantees provided by our approach to context-aware applications.

***Index Terms*—context, mobile, brokers, distributed, Blobseer**

I. 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.

To support this, we designed Context Aware Framework , 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 II 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 III presents details of the implementation, while Section IV presents evaluation scenarios and results illustrating the overall obtained system performance. Section V presents an overview of the main relevant work in the field. We conclude with Section VI.

II. Architecture

A. 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 neighbouring 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, our Context Aware Framework should be able to support *user's mobility and provisioning of data according to his/her 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/her current interests. For example, if a tourist is looking for information about objectives near him/her, he/she 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/she 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 user is always connected to the Internet (a wireless connection might not be available, or in roaming the connection might not come cheap). 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 WiFi).

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, we also analyzed the functional demands coming from a context-aware application. For this we analyzed CAPIM [6], 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 (a) 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/she is currently in; (b) learning, from the previous choices formed by the visitor, what information

s/he is going to be interested in next; (c) 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; (d) deriving location information from sensors which monitor the user environment; (e) 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 persistance, 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 [10] 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 [10] offers an appropriate alternative, as it provides real-time guarantees, large concurrent access guarantees, and support for eventual consistency through an advanced versioning mechanism.

B. Architecture

Based on the identified requirements, we propose the architecture presented in Figure 1.

The architecture includes several components (see Figure 2): Data and Metadata Clients, Brokers, the Metadata Manager, and a Cloud-based storage layer. We next detail each component.

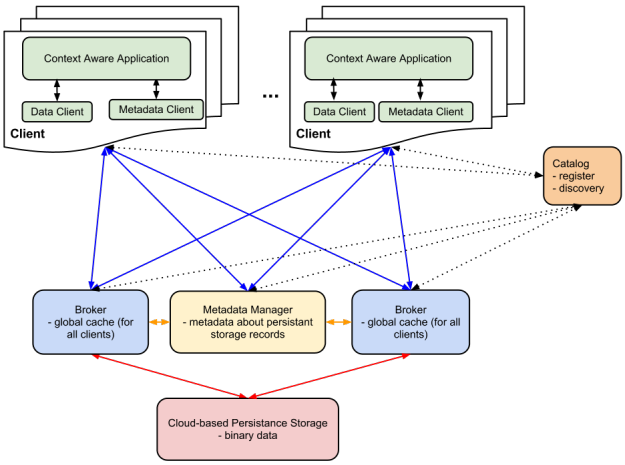

Figure 1. The proposed architecture.

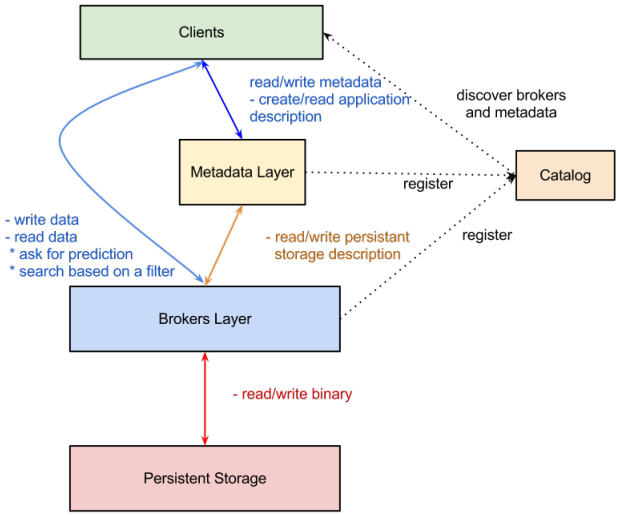

Figure 2. Architecture layers.

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 *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 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 [10] 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.

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 [6] 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.

III. Implementation

We next developed a pilot implementation of the Context Aware Framework architecture previously described.

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, 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 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}:
```

(e.g.,

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

The actual implementation of the *Metadata Manager* uses Mongodb [8], 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 [11].

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.

A. 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.

IV. Experiments, Evaluation and Results

A. Experimental setup

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 [4]. 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 [2], 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 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.

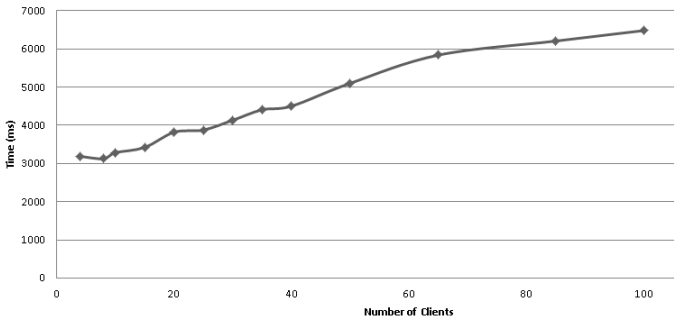

Figure 3. Write Test

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 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 4 and 5 show the results obtained in this case.

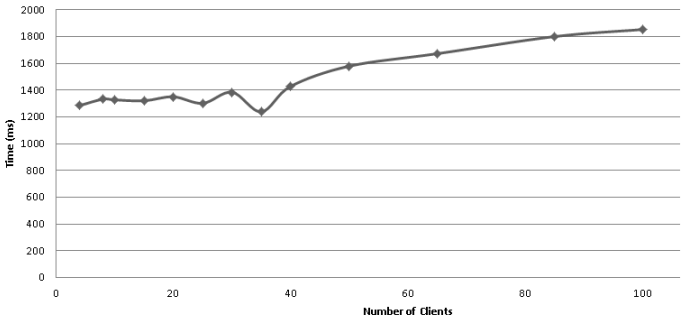

Figure 4. Simple Search

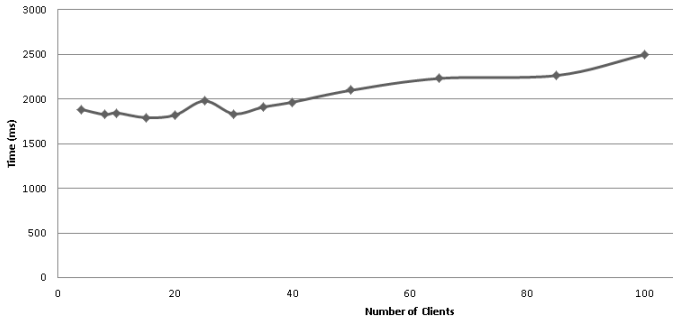

Figure 5. Complex Search

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.

V. 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. Cloud computing provides some of the features needed for these massive data streaming applications. 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 [5] 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 [7]. 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 [9] 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. 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, our platform can also be compared with Google Now [1] and Microsoft On{X} [3]. Google Now [1] 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}.

VI. Conclusions and Further Work

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 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.

VII. Acnowledgment

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 Grid5000/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] Google now. [Accessed February 9th, 2013].

[2] Grid'5000. [Accessed February 9th, 2013].

[3] Microsoft onX. [Accessed February 9th, 2013].

[4] San francisco taxi dataset. http://crawdad.cs.dartmouth.edu/meta.php?name=epfl/mobility.

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

[6] 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, pages 266–272, Washington, DC, USA, 2011. IEEE Computer Society.

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

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

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

[10] B. Nicolae, G. Antoniu, and 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, pages 18–25, New York, NY, USA, 2009. ACM.

[11] D. P. P. PŁŁkknen. Report on scalability of database technologies for entertainment services. http://virtual.vtt.fi/virtual/nextmedia/Deliverables-2011/D1.2.3.3_MUMUMESE_Report%20on%20Scalability%20of%20database%20technologies%20for%20entertainment%20services.pdf, 2012.