Context Storage Using NoSQL

Nuno Santos
Instituto de Telecomunicações
Campus Universitário de Santiago
3810-193 Aveiro - Portugal
Email: nfvs@ua.pt

Oscar M. Pereira
Instituto de Telecomunicações
Campus Universitário de Santiago
3810-193 Aveiro - Portugal
Email: omp@ua.pt

Diogo Gomes
Instituto de Telecomunicações
Campus Universitário de Santiago
3810-193 Aveiro - Portugal
Email: dgomes@ua.pt

***Abstract*—With the ubiquity and pervasiveness of mobile computing, together with the increasing number of social networks, end-users have learned to live and share all kinds of information about themselves. As an example, Facebook reports that it has currently 500 million active users, 200 million of which access its services on mobile systems; moreover, users that access Facebook through mobile applications are twice as active as non-mobile users, and it is used by 200 mobile operators in 60 countries [1]. More specific mobile platforms such as Foursquare, which unlike Facebook only collects location information, reports 6.5 million users worldwide, and also has a mobile presence (both with a web application and iPhone / Android applications) [2]. Context-aware architectures intend to explore this increasing number of context information sources and provide richer, targeted services to end-users, while also taking into account arising privacy issues.**

While multiple context management platform architectures have been devised [3], this paper focuses primarily on Context-Broker-based architectures, such as the ones proposed in the projects Mobilife [4] and C-Cast [5]. More specifically, it focuses on the context management platform XCoA [6]. This platform uses XMPP for its main communication protocol, and publishes context information in a Context-Broker. This context information is provided by Context-Agents, such as mobile terminals, sensor networks and social networks. Due to the nature of the XMPP protocol, the context information is provided in XML form.

This paper proposes the usage of a NoSQL storage system for the purpose of context information storage and retrieval in an XMPP broker-based context platform such as XCoA, together with a full-text searching engine. Through a comparison made through prototypes, the paper clearly demonstrates the advantages of NoSQL storage systems applied to the area of Context Management.

***Index Terms*—Context Management, Knowledge Management, Data Management, NoSQL**

I. Introduction

Mobile computing is becoming ever more ubiquitous, which ultimately has led to an increase in the amount of data related to users and their devices that can be collected to provide better targeted services. This data is referred to as Context Information [7] and consists of all information about a subject (a user, a device, a room, an application, a service) that can be collected and used to provide better services for the same subject or other users. For example, a user's location can be crucial in delivering targeted restaurant recommendations, as well as his personal preferences and restaurant history. This requires a mechanism to properly store, request, manage and analyze context information. In this paper we present a centralized solution built around a component responsible for the distribution and management of context information, commonly referred to as Context Broker (CB). All context information collected or consumed in the Context Management network crosses this component, which has the ability to re-route the information and store a log of all context collected and consumed.

This paper focuses specifically on XCoA [6], an XMPP-based context architecture, which uses the XMPP Publish-Subscribe protocol to handle all context publishing and notification facilities. Context is published to the Context Broker (CB), and Context Consumers (CC) subscribe to specific types of context information and can be notified whenever new context is published. Although the XCoA platform is based on the XMPP Publish-Subscribe protocol, which does not require that all information that passes through it be stored, the ability to store this information provides an important advantage, and enables the creation of a complete context history repository for further analysis.

Current broker-based context management platforms opted for relational databases to handle the storage of context information. Although this has many advantages, as most context management architectures map well to the relational model, the sheer amount of published context information may quickly exhaust a traditional relational database's capacity to respond effectively.

This calls for a storage system that is able to store and handle large quantities of context data efficiently, while still taking into account the relational model of the context management architecture. In this paper we will address the state of the art in Broker based Context Management platforms and in NoSQL storage solutions in section II, followed by a description of the XCoA platform in section III. Next we describe what are the advantages of the usage of a NoSQL solution in the XCoA platform in section IV, and describe how this integration was possible, together with possible deployment architectures and an evaluation of its performance in section V. We end this paper with some final remarks and future directions of the field VI.

II. State-of-the-Art

A. Context Management Platforms

While context acquisition, processing and distribution can follow either a centralized or distributed paradigm, this paper

focuses solely on centralized architectures. This type of architectures have a centralized piece, usually called a Context Broker (CB), which is responsible for handling the relationship between context sources and actuators. Several projects have implemented different architectures for Broker-based context platforms. The Mobilife project [4] was an EU project which implemented such a Broker-based context architecture, where the CB assumed a passive role, being more of a registration and lookup directory. In this platform no context information was persistently stored, as that was assumed to be "almost the same as caching all the data that is routed over a server in the Internet" [4]. No history of past context data was kept, so actuators would only be able to reason about up-to-date context information.

The C-Cast project [5] was built on top of the Mobilife architecture, and provided a way to persistently keep a history of published context information. The CB in this architecture assumed a more active role, managing the relationship between context sources and context consumers and storing context information in a relational database. However, context types were each stored in a different database table, so with every new context type a new database table would have to be created and the data accommodated to it. All communication between the platform entities was handled through XML messaging, and context was stored in a XML-based format called ContextML

B. NoSQL

NoSQL storage systems are a relatively new breed of storage systems that differ significantly from the traditional relational databases. They exchange complex data models and rich query functionalities usually found in relational databases for simpler models, which translates in better performance and distributability, which is key to horizontal scalability. Horizontal scalability is the ability to add more machines to cope with increased data loads, as opposed to vertical scalability which consists in adding more resources to a single machine.

Currently a myriad of NoSQL storage systems exist, distributed across different types and with different goals and features. Within all these different types and goals, one feature seems to be shared among virtually all of them: the "lack of relations" [8]. Virtually among all of them because some NoSQL platforms do provide some primitive mechanisms that allow the existence of relationships between data items. However, this is not their primary goal.

The first so-called NoSQL storage systems arose primarily out of scalability concerns. Google's BigTable [9] and Amazon's Dynamo [10], which are seen as the fore-runners in the development of NoSQL storage systems and seem to inspire many of today's NoSQL applications [8], all handle large, ever-growing amounts of data. These NoSQL storage systems sacrifice features usually found in traditional relational databases, such as ACID properties (Atomicity, Consistency, Isolation, Durability), strong consistency and rich data-query model, in favor of higher scalability, availability and performance.

Most NoSQL storage systems are also schemaless, or schema-free, where no schema is configured or enforced when using the database. Data items with different structures can be mixed and stored, which significantly reduces the complexity of these systems, especially from the point of view of an application developer, as when a data item with a new structure needs to be stored, no changes to the data model are needed.

There are three types of NoSQL storage systems, further discussed in detail: Key/Value Store, Wide Column Store or Column-Oriented, and Document-Oriented storage systems.

C. Key/Value Store

Key-Value Store storage systems are very simple systems, similar to Hash Tables, that store any type of data indexed by a key. Due to its low complexity they are generally very highly performant, but not very flexible. In such system's all data accesses must be made using the given key and there is no way to query data items using any of the contents of the data (unless they are the key).

Amazon Dynamo was one of the first NoSQL storage systems, developed by Amazon only for internal use. It focuses on high reliability and scalability, two crucial requirements at Amazon [10]. Redis is another example of a Key/Value Store; Redis is an open-source Key/Value Store, that typically holds data items in memory, although it can also persist them to disk [11].

D. Wide Column Store

Column-Store, or Column-Oriented, is the name given to storage systems that store information grouped by columns, unlike traditional relational databases which store information grouped by rows. By grouping data by columns instead of rows, all similar data items are physically close to each other. This has benefits when making operations with small sets of columns, but large sets of rows. In traditional relational databases, when updating or accessing a specific column on several rows, this could mean several disk-seeks would need to be performed to find the right columns in each row; with a Column-Oriented database, a single disk-read / write could be sufficient.

These storage systems may not be completely schema-free; there is usually a fixed number of special columns, named Column-Families or Supercolumns, which can then contain several simple columns inside. For example, Cassandra, a decentralized, eventually-consistent Column-Store database developed by Facebook, uses this type of structure, where there is a fixed number of Supercolumns (although they can be changed offline), and each Supercolumn has an infinite number of columns associated with them [12].

E. Document-Oriented

Document-Oriented storage systems are an evolution of Key/Value Store systems; they also store data items indexed by a key, but usually the data items stored are structured documents, more commonly JSON (JavaScript Object Notation)

documents. This allows them to offer a richer set of features on top of those offered by simple Key/Value Store systems. CouchDB an open-source cross-platform document-oriented storage system and a flagship NoSQL solution, stores JSON documents indexed by a Key. In addition CouchDB supports views, transformations of the data items through Map/Reduce functions, that allow any field of a document to become key, therefore enabling document searching by fields other than the initially defined keys [13] [14].

These solutions have an emphasis on distributability, high availability and performance, sacrificing features such as strong consistency for weaker types of consistency. CouchDB, for example, uses Multi-Version Concurrency Control, where documents are never updated; instead, a new version of the document is created, where concurrent read operations can retrieve different versions of a document, although eventually all instances of the database will be in a consistent state.

III. XCoA - XMPP Context Architecture

XCoA is a Broker-based Context Architecture proposal, by D. Gomes et al. [6], built around the XMPP [15] protocol, more specifically XMPP Publish-Subscribe [16]. It built on top of the results of C-Cast, but using the XMPP Publish-Subscribe protocol for communication instead of Web-Services.

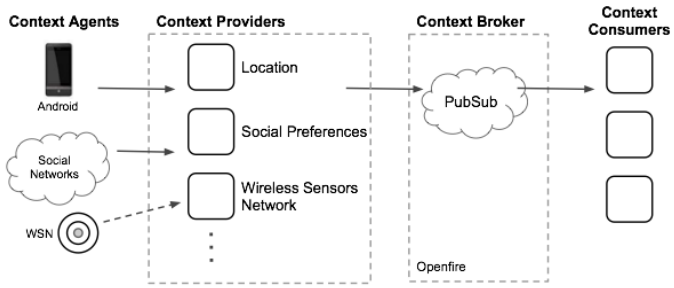

Fig. 1. XCoA Architecture [6]

In this architecture, context information is first collected by Context Agents (CA) such as mobile terminals, social networks, or wireless sensor networks, and aggregated by CPs dealing with specific context scopes such as Location, or Social Preferences. Each CP is responsible to publish the aggregated context information on a CB Publish/Subscribe service which stores this information (History), and sends it to CCs as requested using the PubSub model. All the context information between the CPs, the CB and the CCs is exchanged in XMPP Publish-Subscribe messages. CCs can subscribe to specific Context data, such as Location, and receive notifications when the Providers publish information in the Broker.

Due to the nature of the XMPP protocol, all context information is published in XML form, inside XMPP PubSub Items [16]. CPs can be organized as PubSub Nodes, and context information is published in the respective nodes [6]. CCs can then subscribe specific nodes, for specific context types, receiving only the desired information. Most XMPP servers use a relational databases for persistent storage, and as such store Items as an XML string in the database. Such was also the case with XCoA platform, which used Openfire as an XMPP server and PostgreSQL as a database.

IV. Context Storage using NoSQL

Although in a Publish-Subscribe system it is not necessary to store every published item, in a Context Management Platform it becomes advantageous to do so. Storing every piece of context information provides a comprehensive history of a user's context data, which allows for very powerful features such as context-aware advertising, actuation and environment adaptation according to the user's preferences.

With the need to store every user's full context history comes the problem of handling very large quantities of data. This is a very active research problem, with several proposed solutions such as the so-called NoSQL storage systems, some of which already mentioned in section II.

The information to be published, as previously mentioned, is in XML format. This information can have any structure, and the Platform should be able to efficiently handle context information with any XML structure.

The type of queries used in a Publish-Subscribe system are simple listings of context information, grouped by published Node (E.g. the last 10 Items published on node X) and sorted by publishing date. It should also be possible to retrieve published Items by ID.

This paper, then, proposes the usage of a NoSQL storage system for storing context information with the following main goals in mind: better performance when handling massive ammounts of data, horizontal scalability and availability, and integration with a full-text searching engine. We begin by looking at performance advantages.

A. Performance Advantages

As seen in section III, the XMPP Publish-Subscribe protocol is a good fit for centralized context management platform. CP's, which receive context information from CA's, can publish the context information in the CB, through XMPP PubSub Items. CCs, which subscribe certain context types (through XMPP PubSub Nodes), receive notifications with the context information, as published by the CPs.

Besides real-time notifications from context publication, CCs can ask the CB for published context data. Due to the organization of the context data through XMPP PubSub Nodes, it can retrieve, for example, the last Location Item published, or the last 10 Social Preferences Items.

This makes for two distinct operations, with regards to context publishing. One will be a real-time notification to all the subscribed entities of a context type (PubSub Node), while the other corresponds to a request of context information.

For the first operation, context information storage is largely irrelevant for the completion of the notification procedure. It is theoretically possible, for example, to first notify all subscribed

entities, and postpone the context data persistence to disk (due to the second operation, it may not be desirable).

For the second operation, however, context data will be retrieved from storage. Due to the nature of the context information, it may be acceptable for the information retrieved to miss one context item that's being published in the exact moment the retrieve operation is executed. E.g. when publishing GPS coordinates every 10 seconds, it may be acceptable to, when retrieving the last published GPS coordinates, return the outdated GPS context data (although it will only be, at most, 10 seconds old; hardly meaningfully outdated).

This relaxed persistency constrains allows the tradeoff of strong-consistency features, as present in relational databases, for weaker forms of consistency, when accompanied by performance, reliability and availability gains. A NoSQL storage system will do such a trade.

B. Horizontal Scalability and Availability

Horizontal scalability refers to the ability of scaling a database by adding more machines or nodes, as opposed to vertical scalability which means scaling a database by adding more resources to a single node. One important feature of NoSQL solutions is distributability, which translates in the ability to distribute the database through several nodes, thus scaling the database cluster horizontally.

Together with distributability and horizontal scalability comes increased availability and reliability. Distributing a database through several nodes increases availability and reliability, depending on the distribution approach. Several distribution approaches are possible, such as partitioning data and storing every partition only once, which does not guarantee increased data reliability; however, distributing every partition through two or mode nodes increases data reliability and availability.

C. Full-Text Searching Capabilities

One disadvantage of most NoSQL solutions over relational databases is the inexistence of searching capabilities. Although some NoSQL solutions offer this, most are mostly focused on efficient storage of data and simple retrievals of documents by key.

However, to plug this gap several external full-text searching engines were developed, which can then be integrated into these NoSQL solutions. This provides an important advantage of being able to focus entirely on the NoSQL solution without regards to efficient searching capabilities, and being able to externally plug a searching engine, which usually has the focus on efficient searching. This separation means we can replace one searching engine for a more efficient one, without switching the NoSQL storage system, provided that the searching engine provides integration with the storage system.

With an external searching engine, it would index context information asynchronously, in a passive way, without compromising the correct functioning of the XMPP PubSub protocol. Through replications, one storage node could be in charge of all store/update functionalities, while a replicated node could be in charge of context information indexing and searching. It should be noted that existing relational databases such as PostgreSQL already provide full-text searching capabilities natively, and do not need external components to do it. It is, however, tightly interated inside PostgreSQL, so it is not possible to switch it for a more efficient one, or even to separate it completely.

Using an external searching engine also provides additional scalability benefits, as most of them allow distributability and may be deployed as a cluster, separately from the NoSQL storage cluster, so different scalability requirements for search should not affect the scalability requirements of the storage system, and both can be thought of being deployed independently.

V. Implementation and Results

In order to study the feasibility of storing Context Information in a PubSub XMPP Component, a prototype was implemented.

It was decided to use an external XMPP PubSub component that integrated with an existing XMPP server and offered XMPP PubSub capabilities.

For the selection of a NoSQL solution, both performance advantages, scalability, reliability and indexing / searching capabilities were kept in mind.

CouchDB is a document-oriented database with many important features such as replication capabilities, distributability, allows for horizontal scalability, as well as integration with external document indexing / searching. It is open-source, widely deployed and mature. Other NoSQL solutions such as document-oriented MongoDB were also kept in mind. MongoDB, however, does not offer data durability guarantees in single-node deployments. Wide-column store Cassandra and HBase offer data durability, but their complex data model was deemed incompatible with the XML nature of context information, as it would fit better in a document-oriented database.

The two most popular full-text searching engines are Apache Lucene, mature and widely used [17] and ElasticSearch. In addition, there are already several integrations with NoSQL solutions, such as Cassandra and CouchDB. Since separate indexing of content from a Cassandra dataset is not possible, CouchDB proved to be a good choice.

For the implementation of the external XMPP PubSub component, an existing one was chosen: **Idavoll** [1]. It is implemented in Python and uses the Twisted Framework, an event-driven networking framework. Although most of the protocol was already implemented, some features had to be implemented, such as XMPP XEP-0248: PubSub Collection-Nodes, which supports node hierarchy [18], support for configuring node and subscription options and a Hybrid PostgreSQL/CouchDB storage engine. It was chosen to store only the Items in CouchDB, instead of all the data, because XMPP PubSub Items are not related to other data (its information,

[1] http://github.com/nfvs/idavoll

such as the Node to which it's published, is static, so there's no need for relations between the data). The remaining data (Nodes, Subscriptions, Entities) fits well in a relational model, and is then stored in PostgreSQL.

The context information, contained in PubSub Items in XML form, is stored in CouchDB as a JSON document field, as an XML string. Storing this information inline in the documents, converting it from XML to JSON, was a possibility, but a one-to-one conversion between the two is not possible.

The structure of the documents stored in CouchDB, representing Items, is as follows:

```
{
  "doc_type": "item",
  "item_id": <item_id>,
  "node": <identifier of the node>,
  "publisher": <JabberID of the publishing entity>,
  "date": <Date and time of item publishing>,
  "data": <Item XML String>
}
```

Storing hierarchical information in SQL, as needed by XMPP XEP-0248: PubSub Collection Nodes, provided an additional challenge. In this spec, nodes can be one of two types; either leaf nodes, on which items are published; or collection nodes, which can either contain more collection nodes, or leaf nodes. Subscribers can then subscribe to a single collection node, and receive notifications from all contained leaf-nodes.

Implementing the Nodes tree in SQL can be done in one of two ways: either using Nested-Sets, or Adjacency List [19]. Nested-Sets is a complex model, where each node is numbered according to the order of tree traversal, visiting each node twice and can be implemented without recursion. Adjacency List, on the other hand, is simpler, where each node has a connection to its parent node and although simpler, queries use recursion, which can prove impossible in certain SQL solutions. However, as the database used was PostgreSQL 8.4, which already supports recursive queries [20], the Adjacency List model was chosen.

CouchDB integration with Apache Lucene is then made through couchdb-lucene [2], a java application which indexes CouchDB documents and attachments (including XML attachments).

A. Performance Tests

Performance tests were executed to assess CouchDB's performance against PostgreSQL for the most important operations of the XMPP Publish-Subscribe protocol: Item insertion, which happens every time there is a new published item; Item retrieval, which retrieves a single or multiple Items from the database either by its key or according to other criteria (e.g. the last 5 items published to node X); and Item search, which matches a search string against the full context information contents.

Database insertion of Item publications is not a high performance-demanding operation, as the notification of subscribed entities is not dependent on the Items database. When an Item is published, notifications are issued immediately, and the operation of storing this published Item in the database can even be delayed without impacting the XMPP PubSub protocol. The most important operations are then Item retrieval and Item searching.

These tests were performed within a single node, with the XMPP server, PubSub component, PostgreSQL and CouchDB databases in the same node. The hardware used was a single VMWare virtual machine with an Intel Core 2 Duo CPU @ 2.66Ghz, 512MB of memory and 20GB of disk space. The databases used were PostgreSQL 8.4.1 and CouchDB 1.0.1.

Performance tests were made for Item retrieval by a field other than the key (E.g. by node) and Item searching for datasets of 10K, 50K, 100K, 200K, 500K and 1Million items, both for PostgreSQL and CouchDB. Although in a context management platform the number of publications inserted in the database is always increasing, these datasets already show performance tendencies of both solutions. After hitting the performance limits of the storage system, data partitioning together with new cluster nodes can help mitigate this problem.

It should be noted that CouchDB does not have native support for full-text search queries, which is why Apache Lucene is used in integration with CouchDB. There is currently no supported integration between Lucene and PostgreSQL, so the comparision was made between CouchDB with Lucene and PostgreSQL, Openfire and thus XCoA platform's default database.

Results for Item retrieval and Iteam search are shown in figures 2 and 3.

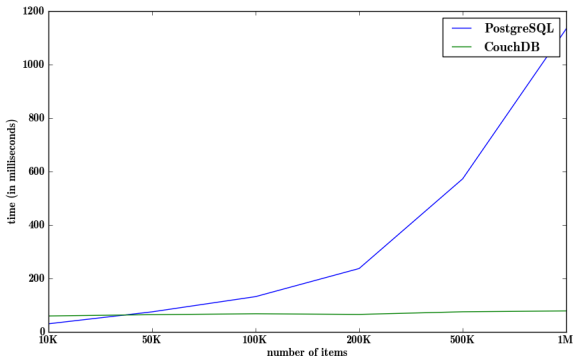

Fig. 2. PostgreSQL vs CouchDB: Item retrieval

Graph 2 shows the performance degradation of Item retrieval when the database increases in the number of Items. Retrieving an Item in PostgreSQL on a 1,000,000-Item database takes on average over 1 second, while in CouchDB takes less than 80 ms. PostgreSQL shows a 98% performance degradation between datasets of 10K and 1M Items, against the 24% degradation observed in CouchDB. Graph 3 shows the performance degradation of Item searching. With the increase of database Items, the performance advantages of the CouchDB

[2]https://github.com/rnewson/couchdb-lucene

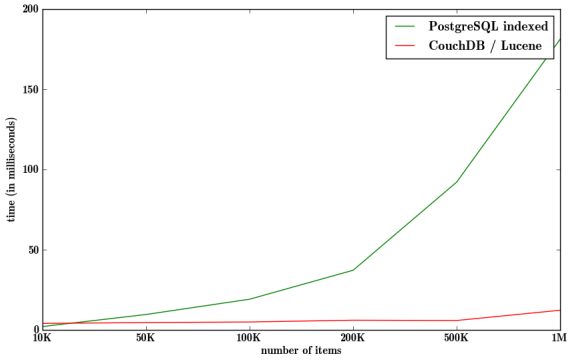

Fig. 3. PostgreSQL vs CouchDB / Lucene: Item searching

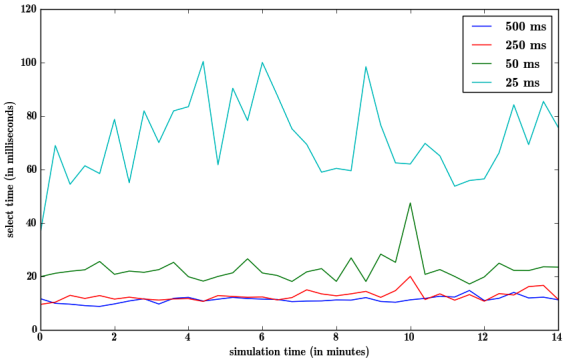

Fig. 5. CouchDB: Item retrieval with simultaneous Item insertions

/ Lucene system become apparent. PostgreSQL shows 98% performance degradation between 10K (about 2ms) and 1M Items (about 181ms), against CouchDB / Lucene's 66% (4ms for 10K, 12ms for 1M Items).

Besides separate retrieval and search query tests, dynamic tests were also made to measure the degradation of each storage system when handling simultaneous insertions and retrievals. For each storage system the Item retrieval time was measured while simultaneously inserting Items every 25ms, 50ms, 250ms and 500ms. The simulation ran for approximately 15m for each database, and for each Insertion rate. The results are shown in 4 and 5.

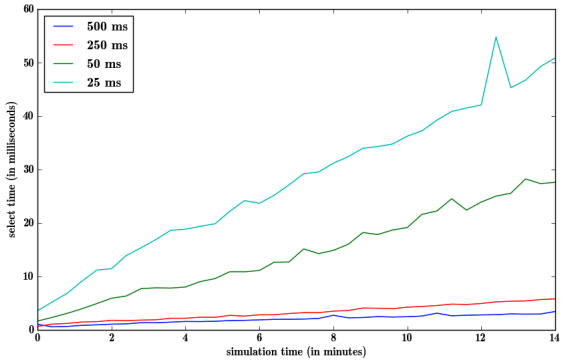

Fig. 4. PostgreSQL: Item retrieval with simultaneous Item insertions

These results show that PostgreSQL's performance degrades linearly with the increase in the number of published Items, while CouchDB's performance remains relatively constant. However, with an insertion rate of one Item every 25ms, CouchDB is initially outperformed by PostgreSQL, although extrapolating from these results we can assume that PostgreSQL's performance will eventually be outperformed by CouchDB. CouchDB also shows very good performance for insertion rates of one Item every 500ms, 250ms and 50ms, with a slight increase in retrieval time for 25ms; however, with a cluster of several CouchDB nodes, it is very unlikely that an insertion rate of one Item every 25ms is ever observed in any single node.

The performance advantages of CouchDB vs. PostgreSQL can be explained by their different approaches; CouchDB focuses on performance, distributability and horizontal scalability, and offers fewer consistency guarantees; PostgreSQL on the other hand offers a richer data model and richer queries, as well as stronger consistency guarantees, while focusing more on vertical scalability and not so much on distributable deployments. These two solutions, with their different focuses, have very different approaches to storing and retrieving data.

VI. Conclusion

The usage of a NoSQL storage system in an XMPP-based Context Architecture provides important performance advantages, and allows for higher availability and reliability, while also scaling horizontally. A NoSQL solution, using CouchDB in particular, proved to be a very good fit for handling large sets of data, such as Context Information. The possibility of integration with the Apache Lucene engine, further allowed searching operations outside the XMPP Publish-Subscribe protocol.

The NoSQL ecosystem, unlike relational databases, is headed towards specialization, so different solutions are headed in different directions, leaving the door open for new players to emerge, and making the ecosystem an exciting and ever-evolving field. In this paper, we addressed the impact and improvements a NoSQL solution can have on a Context Management Platform.

Further work will focus on a fully distributed XMPP Publish-Susbcribe platform, distributing not only the CouchDB nodes but also the PostgreSQL database and the XMPP PubSub component. As in a context management platform the number of context publications always keeps rising, further data partitioning schemes will also be researched, to better handle the always-increasing data load.

REFERENCES

[1] Facebook. Facebook Press Room: Statistics. http://www.facebook.com/press/info.php?statistics.

[2] Foursquare. Foursquare: About. http://foursquare.com/about.

[3] Terry Winograd. Architectures for Context. *Human-Computer Interaction*, 16(2):401–419, 2001.

[4] P Floreen, M Przybilski, P Nurmi, J Koolwaaij, A Tarlano, M Wagner, M Luther, F Bataille, M Boussard, B Mrohs, and Sianlun Lau. Towards a Context Management Framework for MobiLife. *Management*, pages 120–131, 2005.

[5] M Zafar, N Baker, B Moltchanov, J M Gonçalves, S Liaquat, and M Knappmeyer. Context Management Architecture for Future Internet Services. *Applied Sciences*, 2009.

[6] Diogo Gomes, João Gonçalves, Ricardo Santos, and Rui L Aguiar. XMPP based Context Management Architecture. In *IEEE Globecom*, 2010.

[7] Anind K Dey and Gregory D Abowd. Towards a Better Understanding of Context and. In *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 304 – 307. Springer-Verlag London, UK, 1999.

[8] Neal Leavitt. Will NoSQL Databases Live Up to Their Promise? *Computer*, 43(2):12–14, 2010.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable : A Distributed Storage System for Structured Data. *Sports Illustrated*, 26(2):205–218, 2006.

[10] G Decandia, D Hastorun, M Jampani, G Kakulapati, A Lakshman, A Pilchin, S Sivasubramanian, P Vosshall, and W Vogels. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[11] Citrusbyte. Redis: an open source, BSD licensed, advanced key-value store. http://redis.io.

[12] Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[13] The Apache Software Foundation. CouchDB Documentation: Overview. http://couchdb.apache.org/docs/overview.html.

[14] The Apache Software Foundation. CouchDB Wiki: Introduction to CouchDB Views. http://wiki.apache.org/couchdb/Introduction_to_CouchDB_views.

[15] The XMPP Standards Foundation. XMPP. http://xmpp.org/.

[16] Peter Millard, Peter Saint-Andre, and Ralph Meijer. XMPP XEP-0060: Publish-Subscribe. http://xmpp.org/extensions/xep-0060.html.

[17] The Apache Software Foundation. Apache Lucene: Powered By. http://wiki.apache.org/lucene-java/PoweredBy.

[18] Peter Saint-Andre, Ralph Meijer, and Brian Cully. XMPP XEP-0248: PubSub Collection Nodes. http://xmpp.org/extensions/xep-0248.html.

[19] Joe Celko. *Trees and Hierarchies in SQL for Smarties*. Morgan Kaufmann, 2004.

[20] PostgreSQL Global Development Group. PostgreSQL 8.4: Release Notes. http://www.postgresql.org/docs/8.4/static/release-8-4.html.

[21] Meebo.com. couchdb-lounge. http://tilgovi.github.com/couchdb-lounge/.