Control and Cybernetics

vol. **41** (2012) No. 4

Towards effective social network system implementation*

by

Jaroslav Škrabálek, Petr Kunc, Filip Nguyen and Tomáš Pitner

Masaryk University, Faculty of Informatics,
Lab Software Architectures and IS
Botanická 68a,
602 00 Brno, Czech Republic
{skrabalek, xkunc7, xnguyen, tomp} @fi.muni.cz
http://lasaris.fi.muni.cz

Abstract: In this paper we present our latest research in the area of social network system implementation. Both business and technological aspects of social network system development are considered. There are many tools, languages and methods for developing large-size software systems and architectures represented by social network systems. However, no research has been done yet to uncover the reasons behind the selection and usage of such systems in terms of choosing the right architecture and data storage. We describe effective approach to developing specific parts of social network systems with special attention to data layer (using Hadoop, HBase and Apache Cassandra), which forms the foundation of any social network system and is highly demanding for performance and scalability.

Keywords: NoSQL, architecture, social networks, complex event processing

1. Introduction

Social networks – a millennium's first decade phenomenon – have enabled users to connect with people they usually have never seen in person and to live virtual lives, promoted progressive networking, helped people to find a job or just supported gamification (defined as the infusion of game design techniques, game mechanics, and/or game style into anything to solve problems and engage audiences; see Zichermann & Cunningham, 2011) of regular products and services as a very engaging marketing channel.

*Submitted: October 2012; Accepted: November 2012

Size matters. The social networks, unlike common web-based information systems, represent a supreme discipline of software development. No other information system, application or web service could attract millions of users with immensely progressive potential. This unique opportunity cannot be built without a deep requirement analysis including users as main decision makers from the very early phase and, of course, without a careful selection of functionality. For designing a social network system, user-centered approach is particularly crucial. A *platform*, which social network actually is, takes human perspective into account. If customers are satisfied, they are more likely to use new services and recommend the platform to other potential users, which enables the growth of the user basis and provide the foundation for future network. The high number of users indicates high popularity of the platform, which brings more people in. Furthermore, if people use the product, they provide feedback, in particular, they report errors and require new features. Their feedback can lead to significant improvements of the social network and as a result, it can enhance platform as a whole (see Škrabálek et al., 2011).

Architecture is the key. Architecture of the social network system is like a backbone. If it is crooked, the growing potential will never be reached and only a small number of pioneers will use the platform for a limited time and then they will leave it. Scalability and robustness as well as universal analysis with advanced level of flexibility will ensure future enhancement, and eventually will help to completely change the initial intention if users require quite a different functionality. This is, for example, the case of *Takeplace* (see *Takeplace*, 2012) – a digital and mobile event management platform helping event organizers in all event management processes. Thanks to precise definition of core functions followed by an open-minded and foresighted analysis, Takeplace becomes a platform supporting both organizers and community of event attendees to manage any kind of events of any size from small seminars, consultancy meetings with 20—40 participants up to the conferences with hundreds of attendees, or even trade shows, fairs and festivals inviting thousands of people. Soft part of the development such a social network functional requirements, analysis and design is just the first part within the process of social network system development. We need to know *What* to develop but the question *How* follows immediately.

Persistence. Selecting a proper back-end technology is crucial. In the starting phases of the platform adoption by users, it is very easy to handle the demand by standard tools and software approaches one was used to employ in numerous previous projects. The turning point is different in every project but such a time will certainly come and everybody will learn how restrictive our past decisions may be in the case of a social network system. It may even cause the end of the previously well-evolving platform. Therefore, it is indispensable to consider modern persistence tools and frameworks like *Hadoop* from the very beginning since they support handling big data volumes. The non-relational, distributed DBMS HBase and NoSQL database solution Cassandra (described later in the

paper) help developers to keep up with the technological development in time and preserve the direction with respect to contingencies.

Mobile platforms. Such needs are also accentuated by tremendous advent of modern mobile platforms. While the front-end development is simplified thanks to strict usability approaches required by the companies standing behind iOS, Android or Windows Phone 7, the requirements for cloud-friendly back-ends increase continuously. Regarding the existing social networks *Facebook*, *Twitter*, *Instagram* or *Pinterest* the platforms the most people speak about catch around 50 % or more traffic from mobile smartphones and tablets*. Tablets are increasingly appearing and starting to be used as the main working tool. This change of ICT utilization paradigm (see Gartner, 2012) will cause enormous demand for well-developed, not only social network, services and platform solutions in next five years, capable to handle millions of inquiries as well as huge data storages in the backend.

This paper is organized as follows. In this section we have discussed business aspects of social networks. In the second section *Technological demands of social networks and case study* we will introduce one important case study together with its implementation details. The third section is focused on maintainable implementation of social networks using Cassandra NoSQL database. In the fourth section we fill in the gaps in social network implementations by describing monitoring of social networks.

2. Technological demands of social networks and case study

In the domain of social network services, data-oriented architectures and technologies are widely used as those services demand high data throughput. The architectures are designed for heavy loads, concurrent requests, and database can store billions of rows. This section introduces the key features of *Hadoop* and *HBase* (see White, 2009) and describes them in a real application written in Java using HBase as a persistent storage.

2.1. Hadoop and HBase

The use of NoSQL databases means that the data loses relations, developers cannot use the Structured Query Language with joins, triggers, or procedures. Here comes the question why the system architect would want to choose any of NoSQL databases. The main reason is *scalability*. The following story describes how a growing service based on RDBMS usually evolves. Initially, the developers move the system from local environment to the production one with predefined schema, triggers, indexes and in normalized form (3NF or 4NF). As the popularity grows, the number of reads and writes increases. Some caching service is used to improve the read time and the database loses ACID. To improve write

*While Facebook and Twitter register around 40—60 % of mobile accesses, Instagram is a purely mobile social network with 90+ % mobile traffic.

time the components of the database server must be enhanced. New features are added and database schema must be changed – either de-normalized or the query complexity increases. If the popularity grows further, the server has to be more powerful (thus expensive) or some functionality must be omitted (triggers, joins, indexes) (see White, 2009). This is where software framework Hadoop, developed by Apache, is clearly a better solution as it offers automated and linear scaling, automatic partitioning and parallel computing. Hadoop consists of two basic parts:

- MapReduce
- HDFS (Hadoop Distributed File System).

The *MapReduce* model has been introduced by Google. It consists of two phases that both read and write data in a key-value format. The map phase divides the problem into smaller pieces that are then sent by the master node to other distributed nodes in order to be processed. After solving the current problem at distributed nodes, data is sent back to the master node which processes the responses and assembles the solution of the original problem. This model is suitable in situations when the application needs to "write once" and "read many" while traditional RDBMS are designed for frequent data writes or updates. The MapReduce model is also designed to run on commodity hardware so it deals with node dropouts: whenever (in the map phase) the node does not answer in time, the master node just reschedules the problem to other instance. Thus, the master node is the only bottleneck but there can exist more master nodes in the MapReduce model. *HDFS* was designed to store huge files across the network. The default block size is 64 MiB which improves the seek time and increases transfer rates for vast data. Finally *HBase* is a non-relational distributed database based on the Hadoop framework. Tables are automatically distributed in the cluster in the form of *regions*. Each region is a data subset defined by the first row (included), the last row (excluded) and the region identifier. When the size of data grows so much that it cannot be stored on one machine, they are automatically split and distributed (usually using HDFS) across the nodes.

HBase data model The basic HBase data model is inspired by Google's *BigTable*. Tables are in fact four-dimensional persistent sorted maps. The first dimension is a row - any row has a predefined (on the table level) second dimension – column families. Column families can have variable number of columns (third dimension) containing the data. The fourth dimension can also be used: the version – each column can remember x older versions of data stored in the column. The HBase data model is designed to store billions of rows and millions of columns. Data have no relations so joining the tables is impossible. Keys and values are arrays of bytes, so any data can be maintained. A very simple way to obtain certain data is to access the value of the map by specifying table row, column family and column. The only possibility to obtain more rows is to use sequence scanner fetching rows from some interval as the row keys are stored lexically.

2.2. Takeplace: case study of the architecture

The architecture will be demonstrated on a working example of a social network service designed and implemented for the event management platform *Takeplace* (see *Takeplace*, 2012). However, this subsystem can be used in any service. This goal is achieved by using simple interfaces defining the services. Developer should implement a communication layer dealing with remote calls (for example, REST, JSON-RPC or SOAP) or can even call the services directly when using Java. The communication layer must provide a secure user session identifying the current user.

2.2.1. Inner architecture

The system alone consists of three-layer structure (see Fig.1) connected by interfaces. The service layer provides services for external calls and also takes care of basic authorization of operations to be executed. Data access layer retrieves or stores data from/into the database and its main goal is to transform business objects into data structures and vice versa. The last layer provides basic CRUD methods and creates a simple framework for any non-relational databases or cache. *HBase* and *Memcached* are used in this project.

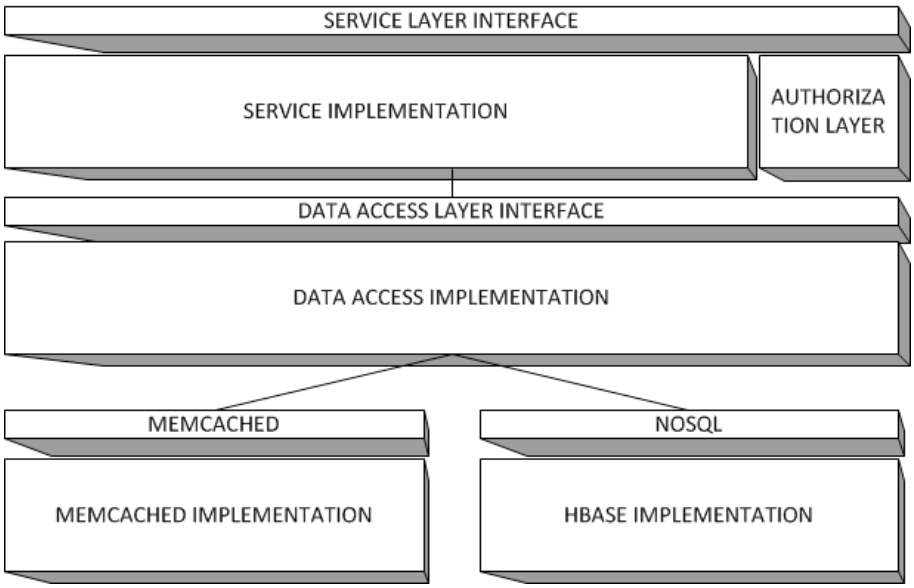

Figure 1. Inner architecture

There are four services available for external calls: *Follow* (managing interactions among users and providing information about the relations), *Wall* (providing interface related to the users' posts, their own walls and news feeds),

Discussion (comments connected to certain post) and *Like* (managing users' favorite posts).

2.2.2. Data model

The data model of the application comprises three tables: *walls*, *entities* and *discussions*. The table *entities* (modeling any user) contains five column families: *followers*, *following* and *blocked* contains user ids in columns. The *news* column family contains ids of posts on news feed (page which shows posts of people the user is following). The last column family *info* contains redundant data about the numbers of followers, following etc. The table *walls* stores posts created by users in the system. The column family *info* stores basic information about the post. In the *text* column family this is only one column containing the text of the post and the *likes* column family contains user ids of people who like the post. The table *discussions* is similar to the table *walls*.

2.2.3. Data storage

While working with non-relational databases the key aspect of design is to choose row identifications as their choice heavily affects performance. The identifications can describe a relation among data and lexical sorting also defines region in which this information is stored. Furthermore, the only way to obtain more rows from the database is the sequence scanner when the rows are sorted lexically. This is why the row identification has to be chosen wisely. For example, table *walls* uses concatenation of the user id and the time of the post, so the posts are grouped by users and then sorted by time from newest to oldest post. Lexically it would be the oldest-to-newest posts so we have to invert bytes in the date format to enable the sequence scanning from the newest posts. Fetching the wall is fast for each user and there is higher probability that it is stored on the same region server. Also each entity can view history of its own posts. There are only weak relations among data. The users (entities) have their posts and they have their comments. These relations are displayed in names of keys and developer is responsible for fetching the correct data.

2.2.4. News feed

The only problem with performance occurs when loading the news feed. These rows of data will be stored across region servers as the row identifier can vary a lot – as the id of any post is assembled from user identification and time – so there would be the need to load a post by post to be displayed from each followed user's wall – or the system could store the posts in each profile creating great redundancy. In this case it is suitable to use memory-caching tool. While sending a new post to the server it is inserted in the cache in a minimized form (also time to live is set) and the link to it is put into the cached news feed to every interested (following) user. Thus, we can obtain a news feed in two cache queries. The first one fetches the list of posts and the second one (batch

query) returns the posts. Memcached is a hash table in random access memory providing fast read/write operations. Once the data gets old or Memcached is full, the expired posts are deleted first and subsequently the least recently used. The backup of the news feed is stored in HBase as the permanent storage.

2.3. Testing the application

The application was tested using *Jakarta JMeter*, *Netbeans Profiler* and private pilot run. The application was tested on three virtual computers, each simulating very old commodity hardware (2 GHz and 1 GB RAM) connected with a 100 Mb/s LAN network. On average, one simple follow invocation took 0.5 ms, fetching the list of one hundred followers and thirty random followers took 2.2 ms (data access time was less than 1 ms). Sending one post to the server consumed 2.58 ms on average and loading wall of 35 posts for a single entity took about 4 ms. Loading time of an entity's news feed was 7 ms. A view of profiler is shown in Fig. 2. On the servers, we got throughput of almost 50 requests per

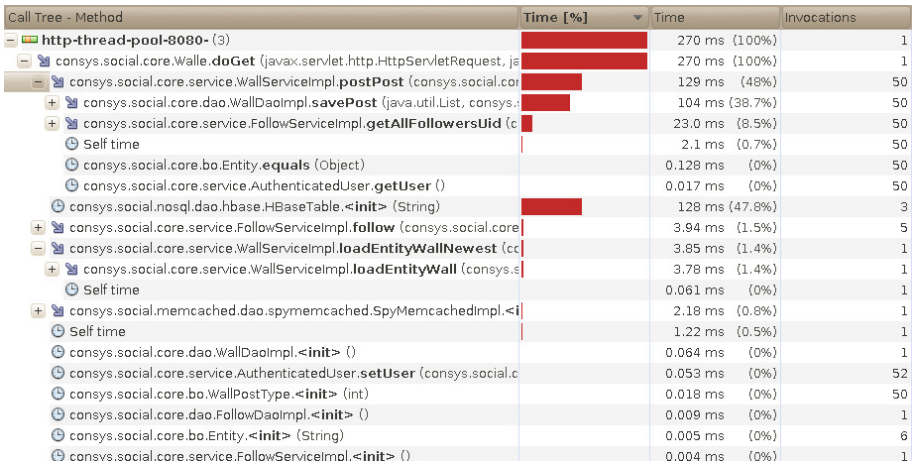

Figure 2. Profiling

second and the median for loading a simple page which performed one follow operation was 290 ms. The most important page's (News Feed) throughput is 44 requests per second and median was 354 ms. Memcached heavily improved the loading time as data load operation of getting all followers improved almost 1000 times (considering only data read time) as the operation needs a single request to RAM. The testing data from the pilot run and load tests look really promising, as the load times are short for common hardware. Next tests we would like to perform will show the results of this system on cloud services, robustness of horizontal scaling under heavy load and we also want to perform tests to compare HBase with MySQL or any other relational database. This software architecture and data model show how the social subsystem can be

implemented using non-relational databases to allow simple horizontal scaling as the data amount grows, high throughput and handling of heavy data loads.

3. Maintainable NoSQL data model using Apache Cassandra

This section shows a simple method for developing Data Layer and Data Access Layer for Social Network written in Java using Cassandra NoSQL database (see Lakshman & Malik, 2010). To get more information about NoSQL data stores see Stonebraker (2010, 2011). To access the Cassandra we use Hector API (Echague et al., 2012) – the most mature API for accessing the Cassandra today. The data layer is a crucial part of any implementation of social network. Such implementation must meet the criteria of

1. maintainability
2. usability
3. transparency
4. compile time checking
5. verification.

The (3) above means that the layer is understandable and comprehensible for new developers, similarly (1) and (2). We are inspired mainly by Fowler (2002) and Larman (1997).

Lastly, the (4) and (5) are something more unique. Java programming language is a language for which an extensive unit testing is very typical. It is mainly due to the fact that pioneers of the Test Driven Development (TDD) come from the Java background (Koskela, 2007). To achieve (4), the developers must use well structured, object oriented, Java API to access their data model. The definition of "well structured" may be disputed, but a programmer with average experience can relatively easily - after a short period of experience with the API - say whether compile time checking may reveal possible problems in this API (see Martin, 2008). If the API does not meet this well structured criterion, the developer has to develop a general implementation. We have come to the conclusion what it means to be the general wrapper. This can be summarized by *Definition of DAL API Generality*: "The Data Access Layer API implementation is said to be general when the implementation need not be changed due to the business requirements".

The Generality is a necessary condition that has to be met for a maintainable NoSQL Data Model.

The (5) is a well known requirement for any piece of software written today. To achieve it, we suggest TDD. In fact, from empirical experience, the TDD works very well with DAL implementations. There are many reasons for that:

- DAL has well defined inputs and outputs.
- Implementation (querying, creating, deleting) is more complicated than testing.
- It is time consuming to test DAL manually, mainly because of the setup/ teardown of test methods. Also note that manual DAL testing is error

prone because the tester will give a lot of false positives.

Justification of reasons why TDD works well with DAL is unfortunately out of the scope of this paper.

Simply by using Java programming language and Cassandra NoSQL database, the solution gets several extra benefits for free:

- openness
- multi-platformity
- easily test driven
- robustness.

Both Java and Cassandra NoSQL database are free of charge and multi-platform – running under JVM. The robustness of Cassandra may be claimed because it was used as Facebook's backing storage for inbox search. Finally, it is possible to embed Cassandra into automated tests.

The goal of this section is to explain how to achieve data model for a social network. A small part of data model for the social network is given. A test driven approach is applied for this model and data access layer (DAO) classes are defined. The DAO layer is a way for a programmer to access the actual data. The rest of the section is devoted to explaining aspects of the DAO layer and TDD to give a detailed insight into the techniques.

3.1. Example

In this section we show how data can be queried. We also present basic objects to access the DAL layer of the social network. Almost every social network should contain the entity *Person*. Assume that Person has two attributes: *name* and *email*. Such a data model is implemented in Java by creating POJO (plain old java object) – object that has no dependencies but Java SDK (standard development kit). Listings for *Person.java* shows a possible person implementation.

Person.java

```
public class Person  implements Serializable {
  private String id;
  private String name;
  private String email;
  //Getters and setters
  ...
}
```

Another part of the data model is the Cassandra Layer. To create an entity in the NoSQL database, Column Families abstraction is used. It is out of the scope of this paper to introduce this concept. The basic idea is that column families are created from source code. This allows good automation and maintainability. Following listing shows such a usage for the Person class:

CassandraBootstraper.java

```
public class CassandraBootstraper {
    public void recreateKeyspace(){
    ...
```

```
      ColumnFamilyDefinition cfd =
        HFactory.createColumnFamilyDefinition(
            keyspace,
            DBConstants.CF_PEOPLE,ComparatorType.UTF8TYPE);
      c.addColumnFamily(cfd);
    }
}
```

Note that when working with the NoSQL database, we are not creating any column definitions. We just define the column family for holding collection of Person data objects. We do not define the schema for attributes (name or email) in any way. Those are added at runtime of the application. The last important piece of code is DAO – Data Access Object. This object is responsible for CRUD operations (Create Read Update Delete) on the entities. There is one DAO for each Entity, hence PersonDAO. Lets take a look at an example of PersonDAO.

PersonDAO.java

```
@Repository
public class PersonDAO extends DAO {
   ...
   public List<Person> findUsersByName(Set<String> names) {
     Rows<String, String, String> result =
          findRows(DBConstants.CF_USERS,
          names, new StringSerializer());
     List<Person> users = parseUsersFromResult(result);
     return users;
  }
}
```

The example shows the read method for getting Person data objects from NoSQL database by their names. The PersonDAO extends the base *DAO* class that contains utility methods like *findRows*. The method *parseUsersFromResult* is a private method of PersonDAO for parsing the *name* and *email* from the database.

3.2. Data layer

While basic Entity-Relationship modeling techniques are well known and studied for years, the NoSQL databases require a different approach. Data in NoSQL database is highly denormalized to gain performance. The data also allow great flexibility in adding new attributes to entities already in the database. When building social network, it is advisable that no proprietary scripts are introduced.

By the Generality Definition we can create a general implementation of *bootstrapping mechanism*. The mechanism is based on the idea that all the test data + schema of the database will be created using the same programming techniques (Java) as is used at runtime. The programmer should use a dedicated class. We name this class *Bootstrapper* in our case. This class has the following responsibilities:

- connect the Cassandra instance

- create schema in empty Cassandra database
- insert test data.

Bootstrapper helps better maintainability because information about schema are versioned in this Bootstrapper class (using Subversion). Bootstrapper also helps testability of code, because unit tests can directly invoke Bootstrappers methods.

To implement a DAO Layer to access Cassandra Database it is advisable to use Hector API because it gives a lot of enterprise level features out of the box. The API introduces a lot of clutter. That is why the best approach to implement DAO is to create base class with general implementation for following actions:

- findRows (columnFamily, keys, resultSerializer) - finds rows with given keys
- findAllRows (columnFamily) - all rows in given family
- findAllObjectRows (columnFamily, objectColumnName) - deserializes the object from given column
- deleteColumn, addColumn, findObjectColumn.

By creating this abstraction the developer can tinker it for a specific social network.

Another important technique to be used for Java+Cassandra DAO layer is *Aspect Oriented Programming* (AOP). We introduced *ErrorHandlingAspect* that is responsible for improving error handling on DAL. The AOP is invaluable in these situations. Reasons stem from the nature of DAL. DAO objects have methods specific for the given entity (Person may have *attachMessage*, which is unique for this entity). It is important that the exception be caught inside of these methods to have bigger picture of the error (parameters, name of the method) and not having to inspect the stack trace. AOP gives us this flexibility. The only thing needed is to declare *AfterThrowing* aspect on all DAO methods. Apart from the error handling, by using AOP we can easily log parameters entering each DAO method.

TDD is a very advisable technique for DAL implementation for applications using NoSQL Database. As the data is highly denormalized and unstructured, it is easy to introduce regression bugs into the code. In-memory Cassandra instance can help to create test suites.

4. Reactive social network monitoring

When studying implementations we came to the conclusion that high volume traffic applications like social networks need very specific approach to their monitoring and also special reactive rules for certain actions happening in the system. This section lays out our findings and latest results from this area. Our results contain both the general framework for deploying such monitoring/reactive infrastructure and best practices to use them.

4.1. Monitoring

The monitoring is a basic activity done in any system being in the production or testing phase. We have found out that in many cases, current monitoring options for data model state changes are not sufficient for such large scale systems because of the following reasons:

- The amount of data generated by social networks is enormous.
- Data changes in social networks may not be deterministic. Furthermore they can be without strict time successions due to a distributed nature of the data model. This is acceptable from the point of view of the user because social networks do not usually contain any time critical information.

Having the data on a *logical node* is very convenient (logical node is, for example, one pluggable component in our layered architecture). The convenience materializes itself in terms of having the right monitoring information in one place for processing. Several problems arise with this approach. A badly written component (*logical node*) may create a bottleneck itself for monitoring information processing. However, the biggest problem is that logical nodes do not respect availability of data over the cluster of data nodes. Data must be moved back and forth between nodes causing serious performance issues.

Data node is, for example, Apache Cassandra cluster node instance. This is the best option from the performance point of view. Having the monitoring information on this node data is not transferred via network nor moved across different file systems for processing. There are, however, certain subtle problems with this approach. No correlation can be done for disparate monitoring information. However, these disparate monitoring information pieces are often related.

4.2. Reactive rules

During the implementation of our case studies – social networks – we were often faced with the necessity to react to specific extreme cases in a dynamic fashion. For example, when a user sends too many notifications per minute, his/her throughput of messages should be limited until his/her payment credibility is checked. Another example might be a situation where continually more respected user has his/her limits relaxed (usual functionality of cloud services from Amazon).

4.3. Methods

At first, we thought that our problems with monitoring and reactive rules are not related. We believed that a combination of *business process execution language* and modification of Apache Cassandra logging system will solve the issues. However, we have found a common solution that solves both problems. We introduced *Complex Event Processing* into the monitoring processing and reactive rules processing. The CEP concept is introduced in Etzion (2011) and Luckham

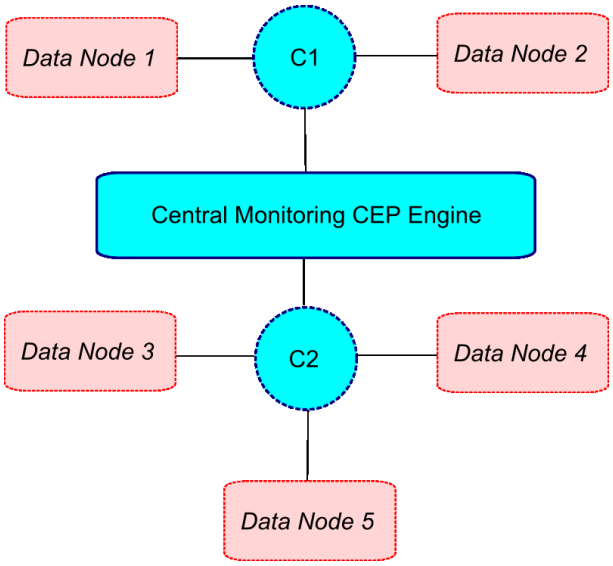

Figure 3. Data monitoring

(2002). The viability of using CEP in distributed environment is supported by several studies in this area, e.g. Lakshmanan, Rabinovitch & Etzioni (2009) and Luckham & Frasca (1998).

CEP solves the dilemma of placing monitoring onto the *logical node* and a *data node*. We choose to put the CEP engines on data nodes and connect them based on logical partitioning done by Cassandra. In this way, different Cassandra nodes are monitored together. The situation is depicted in Fig. 3. In this figure we see five data nodes. A central monitoring CEP engine is used to correlate all monitoring information on high level of abstraction. It does not get all the fine levels of logging messages (e.g. save of particular discussion post in social network) but receives only high-level events that will indicate that some data nodes need a finer analysis. In Fig. 4 we see that central monitoring engine decided that *Data Node 1* and *Data Node 2* should be analyzed together. Analogically, the rest of the data nodes were assigned with *C2* monitoring engine. This architecture allows us to scale monitoring infrastructure very flexibly and automatically.

The issue with reactive rules is solved by publishing events to the service tier of the application from the data tier. Fig. 4 depicts this connection. The connection is in fact the same approach as the solution to the monitoring. We use the monitoring information to gather "hard to get" information without need to go deep into the application logic. In this way we can achieve both scenarios mentioned in the previous section. We can easily monitor how many notifications user sent in a given time window, and through connection to the service layer we can publish this monitoring information. The service layer picks

up this event and acts accordingly by limiting user quota.

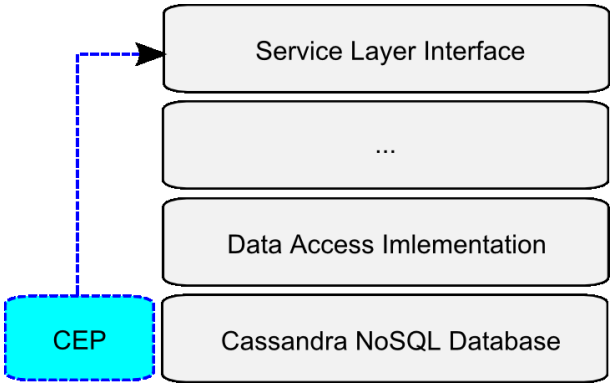

Figure 4. Data layer to Service layer connection

5. Conclusion

In this paper we have presented our current research and development results in the area of social network system development. We showed proper usage of existing frameworks, languages and rationale behind their usage. For a developer, project manager or businessman, being aware of existing tools, their proper implementation and foresight of the future development with close collaboration with users will help to achieve success of the platform and its establishment on the market.

References

Echague, P., McCall, N. et al. (2012) *Hector – A high level Java client for Apache Cassandra*. http://hector–client.github.com/ hector/ build/ html/ index. html. Visited at 11/1/2012.

Etzion, O., Niblett, P. (2011) *Event Processing in Action*. Manning Publications, Shelter Island, USA.

Fowler, M. (2002) *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, USA.

Gartner (2012) Gartner Identifies the Top 10 Strategic Technologies for 2012 http://www.gartner.com/it/page.jsp?id=1826214. Visited at 11/1/2012.

Koskela, L. (2007) *Test Driven: TDD and Acceptance TDD for Java Developers*. Manning Publications, Shelter Island, USA.

Lakshmanan, G. T., Rabinovich, Y. G., Etzion, O. (2009) A stratified approach for supporting high throughput event processing applications. *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, Article 5. ACM, New York, USA.

Lakshman, A., Malik, P. (2010) Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review archive*, **44**, 2, 35-40.

Larman, C. (1997) *Applying UML and Patterns.* 1st edition. Prentice Hall, Boston, USA.

Lin, J., Dyer, C. (2010) Data-intensive text processing with Mapreduce. *Synthesis Lectures on Human Language Technologies*, Morgan and Claypool Publishers, USA.

Luckham, D. (2002) *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Professional, USA.

Luckham, D., Frasca, B. (1998) *Complex Event Processing in Distributed Systems.* Standford University, **28.**

Martin, R. (2008) *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall, Boston, USA.

Stonebraker, M. (2011) Stonebraker on NoSQL and enterprises. *Communications of the ACM*, **54**, 10-11.

Stonebraker, M. (2010) SQL databases vs. NoSQL databases. *Communications of the ACM*, **53**, 10-11.

Škrabálek, J., Tokárová, L., Slabý, J. and Pitner, T. (2011) *Integra–ted Approach in Management and Design of Modern Web-Based Services.* Springer, New York, USA.

Takeplace (2012) An Event Management System http://take-place.com. Visited at 11/1/2012.

White, T. (2009) *Hadoop: The Definitive Guide.* O'Reilly Media, California, USA.

Zichermann, G. and Cunningham, C. (2011) *Gamification by Design: Implementing Game Mechanics in Web and Mobile Apps.* O'Reilly Media, Canada.