Living in the Present: On-the-fly Information Processing in Scalable Web Architectures*

David Eyers
University of Otago
dme@cs.otago.ac.nz

Tobias Freudenreich
Technische Universität Darmstadt
freudenreich@dvs.tu-darmstadt.de

Alessandro Margara
Politecnico di Milano
margara@elet.polimi.it

Sebastian Frischbier
Technische Universität Darmstadt
frischbier@dvs.tu-darmstadt.de

Peter Pietzuch
Imperial College London
prp@doc.ic.ac.uk

Patrick Eugster
Purdue University
p@cs.purdue.edu

ABSTRACT

Today's social web platforms, such as Facebook, Twitter, Google+, and LinkedIn, increasingly have to process large volumes of user-generated data *on the fly*. As the role of such platforms shifts from being portals for largely historic data towards providing platforms for real-time data analytics, we observe that their architectures incrementally move from *storage-centric* designs, based on distributed data management technologies, towards *event-based* models exploiting queueing and stream processing systems.

We believe that it is time to rethink fundamentally the software architecture for social web platforms and base them on a content-based communication model, that is explicitly designed to disseminate and partition incoming request flows on a cluster of servers. A content-based publish/subscribe system thus acts as a scalable and elastic, highly responsive data distribution backbone. By focusing on fresh data, such an architecture can optimize the routing of data to match the topology of the data center, dynamically adapt data flows to alleviate hot spots, and elastically scale to more servers when required by computationally expensive on-the-fly data analytics applications.

1. INTRODUCTION

Social web platforms have transformed how users interact on the Internet. Services such as Facebook, Twitter, Google+, and LinkedIn have had a profound impact on people's lives and have embedded themselves in the daily routines of millions of users. They have become real-time social communication platforms for people to share a rich set of information, including blog posts, photos, videos, chats, and discussions. This has resulted in an explosive growth of the volume of data that must be processed and managed. For example, the number of API calls managed by Twitter has grown from 3 billion per day in April 2010 to 6 billion in September 2010 and the traffic of Twitter.com has grown about 100% in 2010.[1] Currently, the number of Twitter users is growing at a rate of 11 new accounts per second.[2]

A key observation about the evolution of social web platforms is that their focus is shifting from the management of *past* data to the on-the-fly processing of *fresh* data. User-contributed content such as blog posts or photos exhibit the highest popularity when they are new, gradually experiencing lower access rates with age. While in the past, social web platforms were viewed fundamentally as storage repositories for user-contributed content, the emerging emphasis on the dissemination and processing of fresh data means that the low capacity of such systems for on-the-fly information processing is becoming a limitation. Increasingly, services like Twitter are not just providing commentary on events in the real-world, they are themselves changing the real-world.

This exponential increase of new data and need to support new highly responsive services has had significant implications on the software architectures of social web platforms. All major companies have undergone multiple iterations of a complete redesign of their software architectures to satisfy increasing, user-driven demands in terms of scalability, performance and availability. However, often these were *ad hoc* changes to solve specific problems, and did not involve the core communication model of the entire architecture.

Many services in social web platforms require on-the-fly data processing. When new content, such as a blog post, is submitted to a social networking platform, users expect it to be made available immediately to all interested users. In addition, the platform provider wants to perform sophisticated on-the-fly analytical processing of the new data to associate it with matching advertisements [4], to learn new user preferences from the data [1] and to compute aggregated data

*This work was supported by the LOEWE Priority Program Dynamo PLV and the Software-Cluster project EMERGENT funded by the German Federal Ministry of Education and Research under grant no. 01IC10S01.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CloudCP 2012: 2nd International Workshop on Cloud Computing Platforms. Bern, Switzerland
Copyright 2012 ACM 978-1-4503-1161-8 ...$10.00.

[1] http://mehack.com/twitter-by-the-numbers
[2] http://blog.twitter.com/2011/03/numbers.html

for subscribed third parties [15]. Due to the large incoming volume of new data, current best practice is to perform data analytics using periodic batch jobs [11], thus only offering a historic, delayed view on the current data.

The requirement for advanced on-the-fly processing in social web platforms can be witnessed by the use of messaging and stream processing systems that have become part of current designs for such platforms. Facebook, Twitter, Yahoo, and LinkedIn make use of custom-built message delivery infrastructures, such as Kafka [13], Storm,[3] S4 [18], and Hedwig,[4] to support message-based data dissemination between processing servers on-the-fly. However, they still fail in providing an adequate communication infrastructure that is able to optimize routing to the data center topology, dynamically adapt to patterns of network traffic to alleviate hot spots, and to elastically scale in and out for better exploiting network and computational resources.

We believe that it is time to rethink fundamentally the design of scalable architectures for social web platforms, explicitly acknowledging that they should focus on the dissemination, processing and caching of fresh data as it arrives in the system. We argue that a content-based *distributed publish/subscribe infrastructure* [9] should be at the core of such architectures and act as a scalable communication backbone for social web platforms: partitioning data processing and caching. By removing mechanisms for the persistence of past data from the critical path, such an architecture can realize an efficient data-flow model.

We propose to use a publish/subscribe infrastructure that consists of a cluster of *message processing brokers*, which distribute incoming request data. Message brokers transform data and make it available to future requests through efficient in-memory caches with indexing. This reduces the performance requirements imposed on the storage layer because requests are largely satisfied from caches.

To balance the load of requests across servers, routing is adapted based on workload characteristics. The publish/subscribe infrastructure scales out as the workload increases by routing data across a larger set of servers. The communication topology used by the message brokers can adapt to match the physical data center topology. This can minimize network congestion caused by over-subscription of aggregation switches in data centers. Beyond increased scalability, elasticity, and load balancing, our architecture can host and execute third party code directly against fresh data.

This paper is organized as follows. In Section 2, we provide an overview of the evolution of architectures of major social web platforms. Section 3 describes our proposed architecture based on publish/subscribe communication. We discuss related work in Section 4. Concluding remarks are given and future challenges are highlighted in Section 5.

2. SOCIAL WEB PLATFORMS

This section analyzes some of the main social networks, the architectures they adopt, and how they have changed over time. Our analysis highlights three main aspects: *i.* messaging and on-the-fly data analysis are relevant, dominating tasks for a social network; *ii.* they have been the motivations for an evolution of the inner architectures of social networks; and *iii.* some of the existing social web platforms offer disparate services, usually implemented using *ad hoc* components. Integration and data synchronization between these components is a key concern.

2.1 Twitter

Twitter was launched in July 2006 and rapidly gained worldwide popularity, with more than 300 million users in 2011, producing over 2200 tweets and over 18000 queries per second. This is on average: significant spikes have been measured in the past, e.g.,[5] during the Japanese 2011 Tohoku earthquake and tsunami the load increased by 3× to 4×.

Twitter started as a content management platform and underwent several changes to move to the current messaging model, wherein the service tries to update all users with the latest 'tweets' that they have expressed interest in.

Initially, the entire backend was managed using a relational database based on MySQL. Between 2006 and 2010 several optimizations were introduced to meet the dramatically increasing load. They were mainly based on the introduction of several levels of in-memory caches that significantly reduced the delay incurred while processing requests, thus increasing the throughput of the system. Despite these changes, the underlying storage model was still that of relational databases.[6] The limitations of the infrastructure became evident: during the 2010 FIFA World Cup period, Twitter had a high service rejection rate (10%-20%) and a higher average response time[7] than was desired.

The difficulties in scaling the MySQL-based system led to completely replacing it. In late 2010, Twitter launched a new query engine, Earlybird, which is a version of Lucene optimized for low-latency. Earlybird is based on inverted indexes for efficient query processing.[5] This new back-end allowed Twitter to provide users with a more customizable search experience—launched in May 2011—including operators for tag-based, content-based, and context-based (e.g., location, time of tweet) search.[8]

A key performance factor for this new architecture is the ability to rapidly compute indexes when new tweets enter the system, in order to make new tweets available for search as soon as possible. Recently, Twitter announced the acquisition of Storm, a framework that supports continuous, on-the-fly computations over streams of data.[9] Beside tweet indexing, Storm is currently used inside Twitter to distribute other complex computations involving large amounts of streaming data, including trending tags.[10]

All these steps show the increasing importance of on-the-fly data analysis. Not only it is the main reason for all the architectural changes within Twitter, but it also required them to introduce a new product, Storm, explicitly devoted to on-the-fly computation, which is becoming one of the core components of the entire infrastructure. As the system evolves, new functions are being offered to users (e.g., today it is possible to include images into tweets, but we can foresee the possibility of including videos, and filtering multimedia content using customized search operators), which in turn require new processing capabilities. As the scale of the system grows, considering the topology of the process-

[3] http://tech.backtype.com/#post_54482583

[4] https://cwiki.apache.org/ZOOKEEPER/hedwig.html

[5] http://engineering.twitter.com/2010/10/#4731203862532084022

[6] http://blog.evanweaver.com/2009/03/13/qcon-presentation/

[7] http://en.wikipedia.org/wiki/Twitter

[8] http://engineering.twitter.com/2011/05/#6301577286601580570

[9] http://www.slideshare.net/nathanmarz/storm-11164672

[10] http://engineering.twitter.com/2011/08/#1448345121577210449

ing network becomes more important, especially if it spans multiple data centers for replication and high availability.

2.2 Facebook

Facebook is the most popular social web platform with over 800 million active users and 100 billion hits per day.[11] As for Twitter, most of the functionality offered by Facebook requires on-the-fly data analysis: searches, status updates, messages, chat, image processing for tagging, etc.

The available information about Facebook describes it as a federated set of services. Persistence is done using MySQL, Memcached, Cassandra (which was developed by FaceBook, released open source, and is now an Apache project), and Hadoop's HBase. Offline processing is done using Hadoop and Hive. Data such as logging, clicks and feeds transit is performed using Scribe, and data are aggregated and stored in HDFS using Scribe-HDFS, thus allowing extended analysis using MapReduce. Facebook Messages is using its own architecture, which allows for automatic scaling in a cluster of servers [3]. Chat is based on an Epoll server developed in Erlang and accessed using Apache Thrift.[12]

In addition to the key challenges identified for Twitter, integration of different applications and services (including third party extensions) appears to be a key concern in the architecture of Facebook. The need to integrate heterogeneous components, written in different languages and adopting different technologies, can be addressed by adopting event-based infrastructure (e.g. using IBM's MQ product series).

2.3 LinkedIn

LinkedIn, launched in 2003, is the most popular business-related networking site, with 135 million users in 2011.[13]

The architecture of LinkedIn significantly changed over the years—in a similar manner to the architectural changes of Twitter—moving from a database-centric infrastructure based on Oracle and MySQL with cached data, to a messaging architecture.[14] Currently, the back-end of LinkedIn is based on Kafka [13], a distributed messaging system aimed at providing a scalable, low-latency solution for log aggregation and data stream processing. Built on Apache Zookeeper in Scala, Kafka aims at providing a unified infrastructure for both fresh and historical data analysis.[15] Once again, this architecture highlights the increasing need for on-the-fly data processing, as well as the adoption of a core component, Kafka, to deal with it.

2.4 Discussion

Our analysis highlights some key requirements for social networking sites: they need to manage both historical and fresh data. On-the-fly analysis and distribution of data constitutes one of the key tasks for these applications, as exemplified by the processing of tweets and queries in Twitter, and by the query, messaging, and chat services offered by Facebook. Moreover, live and historical data are strictly blended and integrated.

Often, social web platforms offer disparate services to their users. These services share information, reading and modifying common data sets. As the architecture of Facebook underlines, they are usually implemented using different technologies, which further complicates their interaction.

The infrastructures of most social web platforms were originally built around a traditional relational database system. This design choice eventually showed its limitations in terms of scalability, forcing several changes and reorganizations in the architectures of social networking sites as the number of their users grew, putting increasing emphasis on the on-the-fly analysis of streaming data. Often, this led to the adoption of *ad hoc* solutions for on-the-fly processing, like Twitter Storm or LinkedIn Kafka, but these are now becoming the key components of social network platforms.

Despite these changes, current platforms still suffer from several limitations: even when using distributed messaging systems, or stream processing systems. Their data distribution often requires manual optimization to best use the topology of the processing network, and the coordination of software components is often performed using logically centralized systems. We believe that these architectures will not stand for long before requiring reengineering, when compared to the potential of the inherently distributed, extensible and scalable design that we introduce in the next section.

3. RETHINKING THE ARCHITECTURE OF SOCIAL WEB PLATFORMS

We propose an architecture that augments existing online social web platforms with better capabilities for on-the-fly analytical and data processing. Our approach shifts the focus of the architecture from a data storage-centric view to a message-driven one, by exploiting a distributed content-based publish/subscribe system as the core communication mechanism for the entire architecture.

Different features of our architecture will benefit different stakeholders in social web platforms, as explained below. We identify three main types of stakeholders: (1) the *end-users* of the service, e.g. who connect to the platform across the Internet using a web browser or a client application; (2) the *application provider*, who deploys and maintains the social web platform; and (3) *third party developers* who extend the social web platform to provide new features of interest to the end users.

3.1 Contrasting Social Web Platform Architectures

We now introduce our proposed architecture by first illustrating a typical state-of-the-art architecture of a social web platform and then describing our redesigned architecture.

3.1.1 Storage-centric Architecture

Figure 1(a) shows a typical architecture of a social web platform—we refer to this as the *storage-centric architecture*. We show the end-users' connections on the left, which are handled by a set of worker processes. The worker processes are connected to nodes in a caching layer, aimed at mitigating the cost of reading data from persistent storage. The example shown uses `memcached`, which maintains a key-value store along with a cache replacement algorithm. On the right hand side of the subfigure is the cluster of nodes that provide an object store (equivalently, a relational database).

In terms of dataflow, the queries coming from end-users are first distributed among several worker processes. They

[11] http://latimesblogs.latimes.com/technology/2011/09/facebook-f8-media-features.html
[12] http://www.quora.com/What-is-Facebooks-architecture
[13] http://en.wikipedia.org/wiki/LinkedIn
[14] http://hurvitz.org/blog/2008/06/linkedin-architecture
[15] http://blog.linkedin.com/2011/01/open-source-linkedin-kafka/

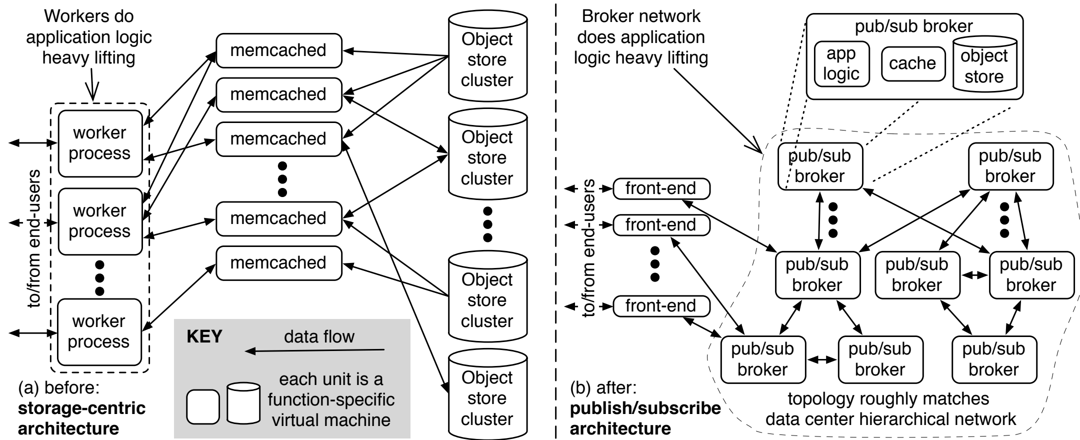

Figure 1: Two architectures for social web platforms: (a) storage centric and (b) publish/subscribe

determine the data items needed to satisfy each query, and contact the `memcached` layer to access them. They typically select the actual `memcached` node(s) to contact by computing a hashing function on the query. On a cache miss, a `memcached` node will be updated with data retrieved from the object store—a comparatively expensive operation, due to the need for disk access.

This illustrates how expensive it is to enable *real-time responsiveness* in such *storage-centric* architectures. Indeed, their *pull-based* nature poses a large computational overhead on the entire infrastructure: conceptually, worker processes have to monitor the database; the cost of this operation is only partially reduced by the use of caches.

3.1.2 Publish/Subscribe Architecture

Figure 1(b) illustrates our suggested architecture. We make use of a distributed content-based publish/subscribe messaging system, enhanced with the possibility to execute the application logic (supplied by the platform itself or third party developers), and linked with a distributed database. This design allows the computations that need to be performed on data to be placed on the path that these data will flow along within the system.

Again end-users are shown on the left, each connected to a lightweight *front-end* process, that receives requests from the end user's client software (e.g. a web browser), and provides responses using appropriate communication protocols and data presentation. To the right of the front-end processes is an interconnected graph of publish/subscribe brokers. Each front-end process forms a network connection to a broker: any broker will do, although those "closest" on the network will be best. As explained below, brokers cooperate to provide a distributed content-based publish/subscribe system. Moreover, each broker contains three other functional components: (1) an engine to execute application logic, (2) a node of a distributed object store, and (3) some cache memory, to accelerate accesses to the object store.

The dataflow through the publish/subscribe architecture is as follows: first, the end-users connect to the front-end processes. These are considerably lighter-weight than the worker processes in the storage-centric architecture, as their computations are largely offloaded into the broker network. Then, the front-end passes the request as a message to the publish/subscribe system. We exploit the routing capabilities offered by the publish/subscribe system to deliver the message to all interested brokers. They will typically include on-the-fly data analysis components working on the information and queries submitted by end-users, e.g., the user profiling tools and trend analyzers adopted by Facebook. If users who express an interest in the published content are connected, their front-end process will be notified by its broker, and new content will be immediately pushed to that user, e.g. by updating the web pages they are viewing.

The content-based publish/subscribe messaging system involves each broker being connected to a number of neighbor brokers, each running on a separate machine. Through their broker, front-end processes can be producers that publish data as messages and/or consumers that subscribe to a subset of these messages by specifying criteria that the message content has to satisfy (e.g. $\mathsf{price} < \$50 \wedge \mathsf{date} = \mathsf{tomorrow} \wedge \mathsf{event} = \texttt{concert}$).

Content-based publish/subscribe systems build an overlay network on top of the physical network. Routing of messages between brokers is done using matching functions that examine the content of messages, instead of on the basis of explicit addressing of hosts on the network [9].

As Figure 1(b) illustrates, the connections formed between brokers should match the data center network's hierarchical topology as closely as possible. In addition to the physical network tree, a second tree root is shown in the figure. This tree does not match the physical network structure, and will cause some inter-broker links to cross levels of the physical network hierarchy. However, doing so will increase the automatic resilience of the overlay network against traffic spikes and equipment failure: the inter-broker overlay network will make best possible use of the underlying physical network links that it has access to.

To provide resilience against failures and overloaded broker nodes, the architecture can use reliable publish/subscribe system techniques, such as multi-path routing of messages, and store-and-forward protocols to ensure that messages successfully pass between successive broker nodes.[16]

The cache for the database is also distributed across the broker nodes and, as in the storage-centric architecture, mitigates the cost of object store disk access.

[16]JMS offers reliability: `http://www.jcp.org/en/jsr/detail?id=914`

The object store and caches are distributed across the set of broker nodes. This allows us to integrate a distributed column store such as Apache Cassandra [14] directly within our architecture, exploiting the same machines that implement the publish/subscribe system.

While our aim is to try to maintain as much of the recent, fresh data as possible in memory across the broker nodes, it will be necessary to make requests of the object store to satisfy queries over older, historical data.

Finally, we plan to adopt our previous work on integrated aggregation [17] within the broker network for providing self-monitoring capabilities to the publish/subscribe system. This allows brokers to collect aggregated information about critical parameters (e.g., number of subscribers connected to each node, number of publishers, message rate, overall traffic), driving possible reactions, such as dynamically changing scale of the broker network. Also, the distributed database could use hints about publisher and subscriber counts to decide where best to persist messages.

3.2 Benefits of our Architecture

Overall, the publish/subscribe architecture has four key advantages, as explored below.

1. Responsiveness. Running application logic directly on broker nodes and pushing data to them allows for responding to new data immediately, and thus fast responses to end-users. Due to the push-based architecture, applications do not have to monitor the database and do expensive queries, but are informed about insertion of fresh data and modification of historic data quickly. For example, users could be provided with a list of posts ordered by the number of their friends who had started typing a comment on those posts.

2. Scalability and Elasticity. The next advantage of our architecture is its design for scalability. If message throughput needs to be increased, one can simply add more machines to the broker network [2]. The publish/subscribe broker network will then transparently integrate the additional nodes (even at runtime) and adapt routing protocols. Because the routing tables used for message dissemination are stored in distributed broker state, scaling up the global system only requires local broker additions. Since there is no need to update global state to increase system scale, the scalability provided is inherently elastic. As brokers are homogeneous, and independent, they can be very rapidly provisioned.

This tremendously benefits application providers: they do not have to think about running their applications in a distributed environment—the broker network abstracts from that, and only needs to be supplied as many machines as needed *at present*, with more machines added on demand. This is particularly useful for cloud-supported elasticity.

3. Load Balancing. When dealing with changing patterns of end-user behavior, e.g. spontaneous traffic spikes caused by flash crowds, systems must have load balancing mechanisms. A distributed publish/subscribe architecture inherently provides load balancing. Routing protocols for distributed publish/subscribe systems usually process messages incrementally at different brokers, as the messages are propagated into the network [9]. Moreover, load balancing is simplified by the content-based nature of the system, which provides for fine-grained classification of message data. End users will not be disappointed by service disruptions, and application providers can use their resources more efficiently.

The load is distributed among brokers more or less evenly, depending on the routing protocol adopted, the topology of the network, and on the actual workload (e.g., how subscriptions are distributed among brokers). We plan to augment the brokers' messages with data that can help the brokers detect and autonomously react to traffic changes more rapidly than possible using only those capabilities already provided by the overlay network. Reactions may include changes to the topology of the broker network, as well as changes in the connections with the front-end processes, to dynamically re-distribute the processing load.

4. Support for Third Party Code. Social web platforms such as Facebook provide end-users with the ability to utilize third party applications, such as calendar add-ons, data management extensions (e.g., for images), and games.

Our architecture provides four significant benefits to stakeholders within social web platforms that allow third party extensions: (1) the application providers retain ownership of their data and do not need to give it away—currently Facebook does not run third party extensions on their servers, instead sending out user data; (2) third-party applications only see the data that they need to see, benefiting privacy and allowing for payment plans based on actual usage; (3) the expressive subscription language alleviates the need for third-party applications to do expensive data filtering; and (4) new applications can be integrated just by adding another message broker, ensuring the preservation of scalability and elasticity even as third-party applications join the platform. Of course hosting third party code on the application provider's servers will require careful security analysis.

Publish/subscribe systems have been shown to work well for system integration, as evidenced from the commercial software systems that apply an Enterprise Service Bus [7]. This should ease the process for third parties to supply code that is to be plugged into the existing dataflow.

4. RELATED WORK

Event processing has been identified as a core element of every complex information infrastructure, realizing a "nervous system" to guide and control the behavior of other sub-systems [16, 5]. Many distributed content-based publish/subscribe systems have been proposed [9]. They place great emphasis on scalability, creating and exploiting overlay networks to process and deliver data with low latency.

A number of dynamic adaptation protocols have been described in the literature, which react to the changes in the network traffic and in the load of brokers by reorganizing the topology of the overlay network [12, 2]. Other solutions propose to move the publishers and subscribers to better distribute the processing load and to reduce the usage of network resources [8]. Most of the results described can be easily adapted to the context of social web platforms, where the brokers are located into one, or few data centers. Our proposal goes a step further, by envisioning a publish/subscribe infrastructure that is able to monitor itself and automatically adapt to changes in the external environment. In [17], we describe ASIA, an integrated aggregation mechanism for distributed publish/subscribe systems: a core building block for enabling self monitoring and adaptation.

Integration of live and historical data is performed in [20, 6], using infrastructure that merges information from event-based systems, and database and data warehouse systems.

Many social networking platforms are now including tools for on-the-fly data analysis in their core infrastructure (e.g., Twitter's Storm, LinkedIn's Kafka—see §2). Moreover, QoS guarantees for low-latency computations are becoming a key concern in the agenda of researchers and practitioners working on data centers and cloud management [10].

Finally, security is a key concern for social web platforms, a concern that becomes even more important when third party applications are integrated within the architecture. Security for publish/subscribe systems has been explored in the past by several works [19, 21].

5. CONCLUSION: OPEN CHALLENGES

In this paper we propose the adoption of a distributed publish/subscribe system as a core communication infrastructure to support social web platforms. We believe this kind of architecture naturally satisfies the increasing demand for fresh data processing and on-the-fly analysis within these platforms. Moreover, it facilitates ease of optimizing the routing strategies to the topology of the processing network inside the data center, to alleviate hot spots, elastically scale in and out, and evenly distribute the load among the available processing nodes. To fully support these features, the publish/subscribe system must implement self-monitoring mechanisms that allow it to detect changes in the external environment and to automatically react to them. We are currently implementing these key features as part of our ASIA middleware [17]. Evaluating our ideas will require us to create, in addition to our publish/subscribe architecture, an instance of an open-source social networking platform comparable to the storage-centric architecture, and to run a realistic user workload against both platforms.

To realize this architectural change, there are some key challenges that need to be addressed. (1) It is necessary to identify the best QoS, load balancing, and adaptation policies that will allow optimization of the use of network and processing resources while satisfying user requirements. (2) In the context of social networks, it is fundamental to clearly specify how to combine fresh and historic data, providing suitable consistency guarantees for current social applications and their future extensions. This includes dealing with replication for data safety and high availability, both inside a single data center and across multiple data centers. (3) As we discussed in Section 2, social networks often need to perform computationally expensive operations on large volumes of data. Therefore, it is important to study how these operations can be split into simple steps and distributed among nodes, to reduce the overall processing latency. We plan to implement this feature by exploiting the in-network aggregation capabilities of our ASIA middleware. (4) Finally, it is important to define suitable security policies to protect user data and to implement mechanisms to enforce them.

6. REFERENCES

[1] E. Agichtein, E. Brill, S. Dumais, and R. Ragno. Learning user interaction models for predicting web search result preferences. In *SIGIR '06*.

[2] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito. Efficient publish/subscribe through a self-organizing broker overlay and its application to SIENA. *Comput. J.*, 50:444–459, July 2007.

[3] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache Hadoop goes realtime at Facebook. In *SIGMOD '11*.

[4] A. Broder. An introduction to online targeted advertising: principles, implementation, controversies. In *IUI '11*.

[5] A. Buchmann, H. Pfohl, S. Appel, T. Freudenreich, S. Frischbier, I. Petrov, and C. Zuber. Event-Driven services: Integrating production, logistics and transportation. In *SOC-LOG'10*.

[6] S. Chandrasekaran. *Query processing over live and archived data streams*. PhD thesis, University of California at Berkeley, 2005. AAI3210530.

[7] D. A. Chappell. *Enterprise service bus*. O'Reilly Media, 2004.

[8] A. K. Y. Cheung and H.-A. Jacobsen. Publisher placement algorithms in content-based publish/subscribe. In *ICDCS '10*.

[9] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131, 2003.

[10] S. Ferretti, V. Ghini, F. Panzieri, M. Pellegrini, and E. Turrini. QoS-aware clouds. In *CLOUD '10*.

[11] M. J. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre. Continuous analytics: Rethinking query processing in a network-effect world. In *CIDR '09*.

[12] M. A. Jaeger, H. Parzyjegla, G. Mühl, and K. Herrmann. Self-organizing broker topologies for publish/subscribe systems. In *SAC '07*.

[13] J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging system for log processing. *NetDB'11*.

[14] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *Oper. Syst. Rev.*, 44:35–40, April 2010.

[15] K. Lerman. Social information processing in news aggregation. *IEEE Internet Computing*, 11:16–28, November 2007.

[16] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[17] A. Margara, S. Frischbier, T. Freudenreich, P. Eugster, D. Eyers, and P. Pietzuch. ASIA: Application-specific integrated aggregation for publish/subscribe systems. Technical report, 2012. http://www.cs.otago.ac.nz/staffpriv/dme/asia/ASIA2011.pdf.

[18] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDMW '10*.

[19] L. Opyrchal and A. Prakash. Secure distribution of events in content-based publish subscribe systems. In *SSYM '01*.

[20] F. Reiss, K. Stockinger, A. Wu, K.and Shoshani, and J. M. Hellerstein. Enabling real-time querying of live and historical stream data. In *SSDBM '07*.

[21] C. Wang, A. Carzaniga, D. Evans, and A. Wolf. Security issues and requirements for internet-scale publish-subscribe systems. In *HICSS '02*.