Designing an Elastic and Scalable Social Network Application

Xavier De Coster, Matthieu Ghilain, Boris Mejías, Peter Van Roy
ICTEAM institute
Université catholique de Louvain
Louvain-la-Neuve, Belgium
{*decoster.xavier,ghilainm*}@*gmail.com* {*boris.mejias,peter.vanroy*}@*uclouvain.be*

***Abstract*—Central server-based social networks can suffer from overloading caused by social trends and make the service momentarily unavailable preventing users to access it when they most want it. Central server-based social networks are not adapted to face rapid growth of data or flash crowds. In this work we present a design for a scalable, elastic and secure Twitter-like social network application, called Bwitter, built on the top of a scalable transactional key/value datastore, such as Beernet or Scalaris. The application runs on a cloud infrastructure and is able to scale its resource usage up and down quickly to avoid overloading and resource wasting. We measure performance, scalability, and elasticity for our prototype and show it performs satisfactorily up to 18 nodes with realistic loads.**

***Keywords*-Scalability; elasticity; cloud application; social network; Twitter; Beernet; Scalaris; key/value store.**

I. Introduction

Social networks are an increasing popular way for people to interact and express themselves. People can now create content and easily share it with other people. The servers of those services can only handle a given number of requests at the same time, so if there are too many requests the server can become overloaded. Social networks thus have to predict the amount of load they will have to face in order to have enough resources at their disposal. Statically allocating resources based on the mean utilisation of the service would lead to a waste during slack periods and overloading during peak periods. Twitter shows the “Fail Whale” graphic whenever overloading occurs [1]. This is a tricky situation as this load is related to many social factors, some of which are impossible to predict. For instance we want to be able to handle the high amount of people sending Christmas or New Year wishes but also reacting to natural disasters. This is why we want to turn towards scalable and elastic solutions, allowing the system to add and remove resources on the fly in order to fit the required load. This work focuses on the design of a social network with elastic and scalable infrastructure: Bwitter, a secure Twitter-like social network built on the transactional key/value store Beernet [2].

This paper summarizes the results of a master’s thesis [3]. Section II defines the basic required operations for a Twitter-like social network. Section III explains why we chose a transactional key/value store, such as Beernet, for implementing Bwitter, and Section IV explains how to run multiple services on top of it. In this section we also discuss some possible improvements for DHTs in order to increase their security and offer a richer application programming interface. Section V presents the application design and Section VI gives our cloud-based architecture. Section VII describes the implementation of our prototype, and Section VIII evaluates its performance (including scalability and elasticity). We then conclude in Section IX.

II. A quick overview of required operations

Bwitter is designed to be a secure social network based on Twitter. Twitter is a microblogging system, and while it looks relatively simple at first sight it hides some complex functionalities. We included almost all of those in Bwitter and added some others. We will only depict the relevant functionalities here that will help us to analyse the design of the system and the differences between a centralised and decentralised architecture.

A. Nomenclature

There are only a few core concepts on which our application is based. A *tweet* is basically a short message with additional meta information. It contains a message up to 140 characters, the author’s username and a timestamp of when it was posted. If the tweet is part of a discussion, it keeps a reference to the tweet it is an answer to and also keeps the references towards tweets that are replies to it. A *user* is anybody who has registered in the system. A few pieces of information about the user are kept in memory by the application, such as her complete name and her password, used for authentication. A *line* is a collection of tweets and users. The owner of the line can define which users he wants to associate with the line. The tweets posted by those users will be displayed in this line. This allows a user to have several lines with different topics and users associated.

B. Basic operations

1) Post a tweet: A user can publish a message by posting a tweet. The application will post the tweet in the lines to which the user is associated. This way all the users following her have the tweet displayed in their line.

2) Retweet a tweet: When a user likes a tweet from an other user she can decide to share it by retweeting it. This will have the effect of "sending" the retweet to all the lines to which the user is associated. The retweet will be displayed in the lines as if the original author posted it but with the retweeter's name indicated.

3) Reply to a tweet: A user can decide to reply to a tweet. This will include a reference to the reply tweet inside the initial tweet. Additionally a reply keeps a reference to the tweet to which it responds. This allows to build the whole conversation tree.

4) Create a line: A user can create additional lines with custom names to regroup specific users.

5) Add and remove users from a line: A user can associate a new user to a line, from then on all the tweets this newly added user posts will be included in the line. A user can also remove a user from a line, she will then not see the tweets of this user in her line anymore and will not receive her new tweets either.

6) Read tweets: A user can read the tweets from a line by packs of 20 tweets. She can also refresh the tweets of a line to retrieve the tweets that have been posted since her last refresh.

III. Why Beernet?

Beernet is a transactional, scalable and elastic peer-to-peer key/value data store built on top of a Distributed Hash Table (DHT) [2][4]. Peers in Beernet are organized in a relaxed Chord-like ring [5] and keep O(log(N)) fingers for routing. This relaxed ring is more fault tolerant than a traditional ring and its robust join and leave algorithm to handle churn make Beernet a good candidate to build an elastic system. Any peer can perform lookup and store operations for any key in O(log(N)), where N is the number of peers in the network. The key distribution is done using a consistent hash function, roughly distributing the load among the peers. These two properties are a strong advantage for scalability of the system compared to solutions like client/server.

Beernet provides transactional storage with strong consistency, using different data abstractions. Fault-tolerance is achieved through symmetric replication, which has several advantages that we will not detail here compared to a leaf-set and successor list replication strategy [6]. In every transaction, a dynamically chosen transaction manager (TM) guarantees that if the transaction is committed, at least the majority of the replicas of an item stores the latest value of the item. A set of replicated TMs guarantees that the transaction does not rely on the survival of the TM leader. Transactions can involve several items. If the transaction is committed, all items are modified. Updates are performed using optimistic locking.

With respect to data abstractions, Beernet provides not only key/value-pairs as in Chord-like networks, but also key/value sets, as in OpenDHT-like networks [7]. The combination of these two abstractions provides more possibilities in order to design and build the database, as we will explain in Section V. Moreover, key/value sets are lock-free in Beernet, providing better performance. We opted for Beernet because of these native data abstractions. But any scalable and elastic key/value store providing transactional storage with strong consistency could be used as well.

IV. Running multiple services on Beernet

Multiple services using the same DHT can conflict with each other. We will now discuss two mechanisms designed to avoid those conflicts.

A. Protecting data with Secrets

Early in the process, we elicited a crucial requirement. The integrity of the data posted by the users on Bwitter must be preserved. A classical mechanism, but not without flaws, is to use a capability-based approach. Data is stored at random generated keys so that other applications and users using Beernet cannot erase others values because they do not know at which keys these values are stored. But in Bwitter, some information must be available for everybody and thus keys must be known by all users, meaning that we cannot use random keys. For example, any user must be able to retrieve the user profile of another user, it must thus know the key at which it is stored. The problem is that Beernet does not allow any form of authentication so key/value pairs are left unprotected, meaning that anybody able to make requests to Beernet can modify or delete any previously stored data.

We make a first and naive assumption that services running on Beernet are bug free and respectful of each other. They thus check at each write operation that nothing else is stored at a given key otherwise they cancel the operation. Thanks to the transactional support of Beernet the check and the write can be done atomically. This way we can avoid race conditions where process A reads, the process B reads, both concluding that there is nothing at a given key and both writing a value leading to the lost of one of the two writes.

This assumption is not realistic and adds complexity to the code of each application running on Beernet. We thus relax it and assume that Beernet is running in a safe environment like the cloud, which implies that no malicious node can be added to Beernet. We allow any application to make requests directly to any Beernet node from the Internet. We designed a mechanism called "secrets" to protect key/value pairs and key/value sets stored on Beernet enriching the existing Beernet API.

Applications can now associate secrets to key/value pairs and key/value sets they store. This secret is not mandatory, if no secret is provided a "public" secret is automatically added. This secret is needed to modify or delete what is stored at the key protected. For instance we could have the following situation. A first request stores at the key *bar* the value *foo* using the secret ASecret, then another request tries to store at key *bar* another value using a secret different from ASecret. Because secrets are different Beernet rejects the last request, which will thus have no effect on the data store. A similar mechanism has been implemented for sets, allowing to dissociate the protection of the set as a whole and the values it contains.

Secrets are implemented in Beernet and have been tested through our Bwitter application. A similar but weaker mechanism is proposed by OpenDHT [7].

B. Dictionaries

At the moment in Beernet, as in all key/value stores we know, there is only one key space. This can cause problems if multiple services use the same key. For instance two services might design their database storing the user profiles at a key equal to the username of a user. This means they can not both have a user with the same username. This problem cannot be solved with the secrets mechanism we proposed. We thus propose to enhance the current Beernet API with multiple dictionaries. A dictionary has a unique name and refers to a key-space in Beernet. A new application can create a dictionary as it starts using Beernet. It can later create new dictionaries at run-time as needed, which allows the developpers to build more efficient and robust implementation. Dictionaries can be efficiently created on the fly in O(log(N)), where N is the number of peers in the Beernet network. Moreover dictionaries do not degrade storing and reading performance of Beernet. If two applications need to share data they just have to use the same dictionary. This has not yet been implemented, but API and algorithms are currently being designed. An open problem is how to avoid malicious applications to access the dictionary of another application.

V. Design Process

We will now present our design choices and explain how we prevent machines hosting popular values from overloading.

A. Main directions

We will start by discussing the main design choices we made for our implementation.

1) Make reads cheap: While designing the construction mechanism of the lines we were faced with the following choice: Either push the information and put the burden on the write, making the "post tweet" operation add a reference to the tweet in the lines of each follower. Or pulling the information and build the lines when a user wants to read them, by fetching all the tweets posted by the users he follows and reordering them. As people do more reads than writes on social networks, based on the assumption that each posted tweet is at least read one time, we opted to make reads cheaper than writes.

2) Do not store full tweets in the lines but references: There is no need to replicate the whole tweet inside each line, as a tweet could be potentially contain a lot of information and should be easy to delete. To delete a tweet the application only has to edit the stored tweet and does not need go through every line that could contain the tweet. When loading the tweet the application can see if it has been deleted or not.

3) Minimise the changes to an object: We want the objects to be as static as possible to enable cache systems. This is why we do not store potentially dynamic information inside the objects but rather have a pointer in them, pointing to a place where we could find the information. For instance, Tweets are only modified when we delete them, if there is a reply to them, the ID of the new child is stored in a separated set.

4) Do not make users load unnecessary things: Loading the whole line each time we want to see the new tweets would result in an unnecessarily high number of messages exchanged and would be highly bandwidth consuming. This is why we decided to cut lines, which in fact are just big sorted set, into subsets, which are sets of x tweets, that can be organised in a linked list fashion, where x is a tunable parameter. This way the user can load tweets in chunks of x tweets. The first subset contains all the references to the tweets posted since the last time the user retrieved the line, it can thus be much larger than x tweets, it is not a problem as users generally want to check all the new tweets when they consult a line. The cutting is then done as follows: the application removes the x oldest references from the first set, posts them in an new subset and repeats the operation until the loaded first set is smaller than x.

5) Retrieve tweets in order: Due to the cutting mechanism and delays in the network we can not be sure that each reference contained in a subset is strictly newer than the references stored in the next subset. So we also retrieve the tweet references from this one and only select the first 20 newest references before fetching the tweets.

6) Filter the references: When a user is dissociated from a line we do not want our application to still display the tweets he posted previously. We decided not to scan the whole line to remove all the references added by this user, but rather remove the user from the list of the users associated with the line and filter the references-based on this list before fetching the corresponding tweets.

7) Only encrypt sensitive data: Most of the data in Twitter is not private so there would be no point in encrypting it. Only the sensitive data such as the password of the users should be protected by encryption when stored.

8) Modularity: Even if our whole design and architecture relies on the features and API offered by Beernet it is always better to be modular and to define clear interfaces so we can replace a whole layer by an other easily. For instance any other DHT could easily be used, provided it supports the same data abstractions or they can be simulated.

B. Improving overall performance by adding a cache

1) The popular value problem: Given the properties of the DHT, a key/value pair is mapped to a node or f nodes, where f is the replication factor, depending of the redundancy level desired. This implies that if a key is frequently requested, the nodes responsible for it can be overloaded while the rest of the network is mostly idle and adding additional machines is not going to improve the situation. It is not uncommon on Twitter to have wildly popular tweets that are retweeted by thousands of users. In the worst case the retweets can be seen as an exponential phenomenon as all the users following the retweeter are susceptible to retweet it too [8].

2) Use an application cache as solution: Adding nodes will not solve the problem, because the number of nodes responsible for a key/value pair will not change. In order to reduce this number of requests we have decided to add a cache with a LRU replacement strategy at the application level. This solves the retweet problem because now the application, which is in charge of several users, will have in its cache the tweet as soon as one of its user reads the popular tweet. This tweet will stay in the cache because the users frequently make requests to read it. This way we will reduce the load put on the nodes responsible for the tweet.

We now have to take into account that values are not immutable, they can be deleted and modified. A naive solution would be to do active pulling to Beernet to detect changes to the key/value pair stored in the cache. This would be quite inefficient as there are several values, like tweets, that almost never change. In order to avoid pulling we need a mechanism that warns us when a change is done to a key/value pair stored in the cache. Beernet, as described in [2], allows an application to register to a key/value pair and to receive a notification when this value is updated. Our application cache will thus register to each key/value pair that it actually holds and when it receives a notification from Beernet indicating that a pair has been updated it will update its corresponding replicas. This mechanism has the big advantage of removing unnecessary requests. Notifications are asynchronous, so the replicas in the cache can have different values at a given moment, leading to an eventual consistency model for the reads. On the other hand writes do not go through the cache but directly to Beernet, this allows to keep strong consistency for the writes inside Beernet. This is an acceptable trade off as we do not need strong consistency for reads inside a social network.

VI. Architecture

Bwitter is designed as a cloud application in which both the Beernet and Bwitter nodes run on a cloud infrastructure and the users are purely clients. We can thus easily add or remove Bwitter and Beernet nodes to meet the demand,

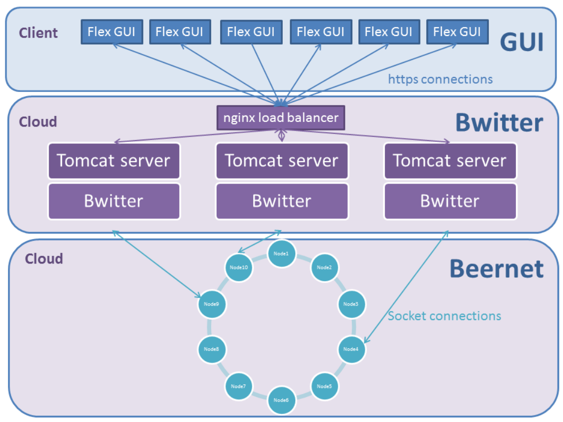

Figure 1. Architecture of the Bwitter social network application

increasing the efficiency of the network. A Bwitter node is a machine running Bwitter but generally also a Beernet node. This solution also allows us to keep a stable DHT as nodes are not subject to high churn as it was the case in the first architecture we presented. The Beernet layer is monitored in order to detect flash crowds and Beernet nodes are added and removed on the fly to meet the demand. We were not able to compare our system with the current Twitter architecure due to the lack of official documentation. But we know that Twitter is centralized, being able to handle only a limited number of concurrent request.

Our application consists of three loosely coupled layers. From top to bottom: the Graphic User Interface (GUI), the Bwitter layer which implements the operations described in Section II and finally the Beernet layer. The overall architecture is very modular and each layer can be changed assuming it respects the API of the layer above. The Beernet layer could be replaced by any key/value store with similar properties (in particular, with transactions and strong consistency). We recall that the data store must provide read/write operations on values and sets as well as implementing the secrets we described before.

The intermediate layer, also running on the cloud, is the core of Bwitter. It communicates both with Beernet and the GUIs. This layer can be put on the same machine as a Beernet node or on another machine. Normally there should be less Bwitter nodes than Beernet nodes. One Bwitter node is associated to a Beernet node but can be relinked to another Beernet node if it goes down. Each Bwitter node should be connected to a different Beernet node in order to share the load. In practice the Bwitter nodes are not accessible directly. They are accessed through a fast and transparent reverse proxy that splits the load between Bwitter nodes.

The top layer is the GUI, which runs on the client nodes and connects to a Bwitter node using a secure connection channel that guarantees the authenticity of the Bwitter node and encrypts all the communications between the GUI and the Bwitter node. Multiple GUI modules can connect to the same Bwitter node.

A. *Elasticity*

We previously explained that to prevent the Fail Whale error, the system needs to *scale up* to allocate more resources to be able to answer an increase in user requests. Once the load of the system gets back to normal, the system needs to *scale down* to release unused resources. We briefly explain how a ring-based key/value store can handle elasticity in terms of data management.

1) Scale up: When a node j joins the ring in between peers i and k, it takes over part of the responsibility of its successor, more specifically all keys from i to j. Therefore, data migration is needed from peer k to peer j. The migration involves not only the data associated to keys in the range $]i, j]$, but also the replicated items symmetrically matching the range. Other NoSQL databases such as HBase [9] do not trigger any data migration upon adding new nodes to the system, showing better performance scaling up.

2) Scale down: There are two ways of removing nodes from the system: by *gently leaving* or by *failing*. It is very reasonable to consider gentle leaves in cloud environments, because the system explicitly decides to reduce the size of the system. In such case, it is assumed that the leaving peer j has time enough to migrate all its data to its successor, which becomes the new responsible for the key range $]i, j]$, being i the predecessor.

VII. Implementation

We implemented Bwitter using the cloud-based architecture of Figure 1. Source code is available at [10]. We made implementations both using Beernet [2] and Scalaris [11]. The architecture has three main layers: the GUI layer, the Bwitter layer, and the DHT layer. The GUI layer is implemented as a Rich Internet Application (RIA) using the Adobe Flex technology. The DHT layer is implemented using Beernet, built in Mozart v1.3.2 [12] enhanced with the secret mechanism. Beernet is accessible by the Bwitter layer through a socket API.

The Bwitter layer is connected to the DHT layer using sockets to communicate with an Oz agent controlling Beernet. The Bwitter layer is connected to the GUI layer with a Tomcat 7.0 application server using Java servlets from Java EE. The Bwitter nodes are accessible remotely via an http API that conforms to REST. The Tomcat servers are accessed indirectly through a reverse proxy server, in this case nginx. This nginx server is in charge of serving static content as well as doing load balancing for the Tomcat servers. This load balancing is performed so that messages of the same session are always mapped to the same Tomcat server. This is necessary as authentication is needed to perform some of the Bwitter operations and we did not want to share the state of the user sessions between the Bwitter nodes for performance reasons. The connection to the Web-based API is performed using https to meet the secure channel requirement of our architecture.

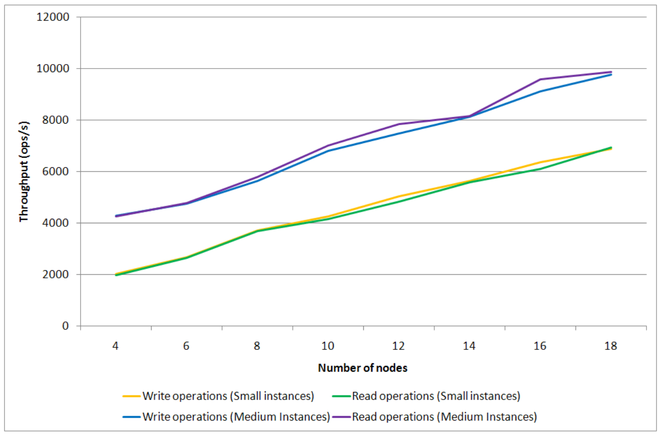

Figure 2. Scalability of the Scalaris transactional key/value store

VIII. Evaluation

We evaluated a prototype implemented with Scalaris v0.3 running on Amazon EC2 with up to 20 compute nodes. Note that we used Scalaris for the evaluation instead of Beernet, for technical reasons unrelated to Bwitter. This section summarizes our most important results; many more measurements and details can be found in [3]. Scalaris and Beernet both have very similar architecture and functionality: both provide a scalable transactional key/value store implemented on top of a replicated DHT and both use Paxos consensus for the transaction commit [2][11]. Since Scalaris underlies our Bwitter prototype (each Bwitter tweet requires many Scalaris operations), we first verified the performance and scalability of Scalaris. Figure 2 shows throughput for 20000 operations (reads or writes) as the number of compute nodes increases. This clearly shows that Scalaris is scalable for both reads and writes, on both Small and Medium size compute node instances in Amazon EC2.

For the Bwitter tests, we use one Large node for the dispatcher and many Small nodes for the Bwitter application. We simulated a network with two kinds of users, “Stars” and “Fans’, where Stars are followed by many Fans. We simulated two kinds of network: a Light network with 4000 users and 25 followers per user (each user follows 0.625% of the network) and a Heavy network with 2000 users and 50 followers per user (each user follows 2.5% of the network). Remark that both Light and Heavy networks have greater connectivity between users than the actual Twitter system, so that we can safely assume they are realistic loads.

Figure 3 shows aggregate throughput (number of successful operations per second) as a function of number of nodes. Here, an “operation” is defined in terms of what users do:

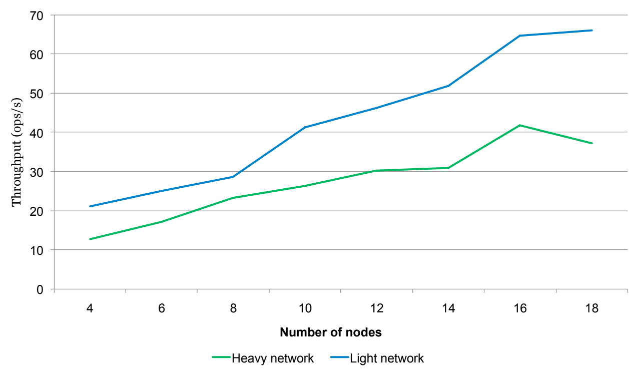

Figure 3. Scalability of the Bwitter application implemented with Scaliris

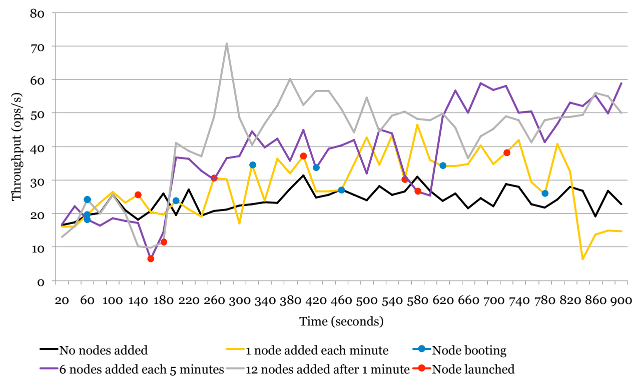

Figure 4. Elasticity of the Bwitter application implemented with Scalaris

it is either posting a tweet (20% of operations) or reading a set of recent tweets (reading all unread tweets counts as one operation; on average 20 tweets are read in one operation) (80% of operations). This means that Bwitter handles 66 operations/second with 18 nodes, which is slightly more than 1000 read/writes of individual tweets per second, in a network with 4000 users. Up to 18 nodes, the number of operations per second increases linearly with number of nodes for both Heavy and Light networks.

Figure 4 shows the elasticity behavior over a period of 15 minutes with four elasticity strategies, i.e., four ways of adding nodes to face increasing load. The black (lowest, almost horizontal) curve gives the baseline (no nodes added). The yellow (intermediate) curve shows the effect of adding one node every minute: the graph shows that this is not a good strategy. The best strategies are the gray and violet ones (highest throughput), in which larger numbers of nodes are added less frequently.

IX. Conclusion

The goal of Bwitter was to build a Twitter-like social network that is able to withstand flash crowds by using an elastic and scalable architecture. We used a scalable transactional key/value store, namely Beernet or Scalaris, as the data storage. We built an architecture on top of this store that is able to handle users with large numbers of followers and users following a large number of other users. We avoid overloading single nodes because we do not rely on any global keys and we use a cache to avoid the retweet problem. Scalability and elasticity tests performed on Amazon EC2 give encouraging results up to 18 nodes with realistic loads. During the implementation we came across two potentially important improvements for key/value stores, namely duplicating the key space using multiple dictionaries and protecting data via secrets (a form of capability). Secrets are now implemented in Beernet.

References

[1] Y. Lu, “What is Fail Whale?” www.whatisfailwhale.info, 2009.

[2] B. Mejías and P. Van Roy, “Beernet: Building self-managing decentralized systems with replicated transactional storage,” *IJARAS: International Journal of Adaptive, Resilient, and Autonomic Systems*, vol. 1, no. 3, pp. 1–24, Jul.-Sep. 2010.

[3] X. De Coster and M. Ghilain, “Designing an Elastic and Scalable Social Network Application,” pldc.info.ucl.ac.be, Programming Languages and Distributed Computing (PLDC) Research Group, Université catholique de Louvain, Tech. Rep., Aug. 2011.

[4] B. Mejías, “Beernet: pbeer-to-pbeer network, version 0.9,” beernet.info.ucl.ac.be, 2011.

[5] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” in *Proceedings of the 2001 ACM SIGCOMM Conference*, 2001, pp. 149–160.

[6] A. Ghodsi, “Distributed k-ary system: Algorithms for distributed hash tables,” Ph.D. dissertation, KTH — Royal Institute of Technology, Stockholm, Sweden, Dec. 2006.

[7] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, “OpenDHT: A public DHT service and its uses,” citeseer.ist.psu.edu/rhea05opendht.html, 2005.

[8] D. Boyd, S. Golder, and G. Lotan, “Tweet, tweet, retweet: Conversational aspects of retweeting on Twitter,” in *Hawaii International Conference on System Sciences*, 2010, pp. 1–10.

[9] Apache, “HBase,” hbase.apache.org, 2011.

[10] X. De Coster and M. Ghilain, “Bwitter source code,” www.info.ucl.ac.be/~pvr/BwitterSources.zip, Aug. 2011.

[11] T. Schütt, F. Schintke, and A. Reinefeld, “Scalaris: reliable transactional p2p key/value store,” in *ERLANG ’08: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*. New York, NY, USA: ACM, 2008, pp. 41–48.

[12] Mozart Consortium, “Mozart Programming System,” www.mozart-oz.org, 2011.