The Little Engine(s) That Could: Scaling Online Social Networks

Josep M. Pujol, Vijay Erramilli, *Member, IEEE*, Georgos Siganos, Xiaoyuan Yang, Nikolaos Laoutaris, *Member, IEEE*, Parminder Chhabra, and Pablo Rodriguez, *Member, IEEE*

***Abstract*—The difficulty of partitioning social graphs has introduced new system design challenges for scaling of online social networks (OSNs). Vertical scaling by resorting to full replication can be a costly proposition. Scaling horizontally by partitioning and distributing data among multiple servers using, for e.g., distributed hash tables (DHTs), can suffer from expensive interserver communication. Such challenges have often caused costly rearchitecting efforts for popular OSNs like Twitter and Facebook. We design, implement, and evaluate SPAR, a Social Partitioning and Replication middleware that mediates transparently between the application and the database layer of an OSN. SPAR leverages the underlying social graph structure in order to minimize the required replication overhead for ensuring that users have their neighbors' data colocated in the same machine. The gains from this are multifold: Application developers can assume local semantics, i.e., develop as they would for a single machine; scalability is achieved by adding commodity machines with low memory and network I/O requirements; and N+K redundancy is achieved at a fraction of the cost. We provide a complete system design, extensive evaluation based on datasets from Twitter, Orkut, and Facebook, and a working implementation. We show that SPAR incurs minimum overhead, can help a well-known Twitter clone reach Twitter's scale without changing a line of its application logic, and achieves higher throughput than Cassandra, a popular key-value store database.**

***Index Terms*—Algorithms, distributed systems, online social networks (OSNs), scaling.**

I. Introduction

THERE has been an unprecedented increase in the use of online social networks (OSNs). The most popular OSNs attract hundreds of millions of users (e.g., Facebook [19]), deliver status updates at very high rates (e.g., Twitter [7]), and distribute user-generated content (UGC) at a global scale (e.g., YouTube). OSNs differ from traditional Web applications on multiple fronts: They handle highly personalized content, encounter nontraditional workloads [11], [34], but most importantly, deal with highly interconnected data due to the presence of strong community structure among their users [12], [27], [30], [31]. All these factors create new challenges for the maintenance, management, and scaling of OSN systems.

The focus of this paper is on scalability. Scaling real systems is hard as it is, but the problem is particularly acute for OSNs due to their astounding growth rate. Twitter, for instance, grew by 1382% between February and March 2009 [28] and was thus forced to redesign and reimplement its architecture several times in order to keep up with the demand. Other OSNs that have failed to do so no longer exist [2].

A natural solution to cope with high demand is to upgrade existing hardware. Such *vertical scaling*, however, is costly since high-performance clusters and even middle-tier servers remain expensive. In some cases, vertical scaling can be technically *infeasible*; Facebook requires multiple hundreds of terabytes of memory across thousands of machines [3]. A more cost-effective approach is to rely on *horizontal scaling* by using a high number of cheap commodity servers and partitioning the work among them. The advent of cloud computing systems like Amazon EC2 has streamlined horizontal scaling by providing the ability to lease virtual machines (VMs) dynamically.

Cloud-based horizontal scaling has aided scaling of *traditional* Web applications. The application front end and logic layer are stateless and can be deployed on independent commodity servers to meet current demand. A similar strategy can be employed for the back-end databases (DB) layer—as long as the corresponding data can be partitioned into independent components, and these components can be stored on independent VMs running relational database management systems (RDBMs) or key-value stores like Cassandra, etc.

In the case of OSNs, the existence of social communities [27], [30], [31] hinders the partitioning of the back end into clean, disjoint components that can be hosted on servers [21], [20]. The problem is caused by users that belong to more than one community. We can place such users on the same server that contains most of their neighbors, however this can lead to high volumes of interserver traffic in order to resolve queries from the users' neighbors from other communities. The problem becomes particularly acute under random partitioning that is often used in practice [19], [22]. On the other hand, replicating user profiles on multiple or all the servers eliminates the interserver traffic, but increases the replication overhead. This impacts negatively on multiple fronts, including the query execution times (due to much larger database tables), network traffic for propagating updates, and the need to maintain consistency across many replicas. Scalability for OSNs is a difficult beast to tame.

Manuscript received February 23, 2011; revised August 15, 2011; accepted October 30, 2011; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor J. C. S. Lui. Date of publication March 22, 2012; date of current version August 14, 2012.

J. M. Pujol was with Telefonica Research, Barcelona 08019, Spain. He is now with 3scale, Barcelona 08018, Spain.

V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, and P. Rodriguez are with Telefonica Research, Barcelona 08019, Spain (e-mail: vijay@tid.es; georgos@tid.es; yxiao@tid.es; nikos@tid.es; pablorr@tid.es).

P. Chhabra was with Telefonica Research, Barcelona 08019, Spain.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2012.2188815

II. Our Contribution—SPAR

The main contribution of this work is the design, implementation, and extensive evaluation of SPAR, a Social Partitioning And Replication middleware for online social network applications.

A. What Does SPAR Do?

Solves the Designer's Dilemma for Young OSNs: A young OSN is often faced with the Designer's Dilemma: "Should it commit its scarce developer resources toward adding features, or should it first ensure that it has a highly scalable system in place that can handle the subsequent high volumes of traffic?" Choosing the first option can lead to "death-by-success"—users come attracted to features, but the infrastructure is not there to provide adequate QoS, leading to frustrated users leaving. That was the story behind the demise of Friendster [2].

On the other hand, starting with a highly scalable system, similar to the ones that power-established OSNs like Facebook, requires devoting scarce developer resources to complicated distributed programming and management issues. This comes at the cost of building features that attract users in the first place.

SPAR enables *transparent scaling* for an OSN by ensuring that all relevant data for a user stays on a machine, thereby maintaining *local semantics* at the data level and letting all queries be resolved locally on that machine. This creates the illusion of the system running on a single centralized server, simplifying the programming for developers and enabling them to focus on developing features.

Avoids Performance Bottlenecks in Established OSNs: By enforcing local data semantics, SPAR avoids the potential performance problems of having to query multiple servers across a network. For instance, random partitioning solutions used by established OSNs like Twitter or Facebook split data across thousands of database servers, which are then queried with *multiget* requests to fetch a user's neighbors data (e.g., all the friends' tweets). This can result in unexpected response times, determined by the latency of the worse server, and can be particularly acute under heavy datacenter loads, where sudden network delays or network congestion can cause performance problems.

In addition to potential network problems, individual servers could also suffer performance problems (e.g., disk I/O or CPU bottlenecks) and drive down the performance of the system. For instance, servers could become CPU-bounded as they need to handle a larger number of query requests from other servers. When a server's CPU is bound, adding more servers does not help serve more requests [1]. Using more servers decreases the bandwidth per server, however it does not decrease the number of requests per server, which means that CPU usage stays roughly the same. SPAR reduces the impact of such multiget operations as relevant data is kept local, avoiding potential network and server bottlenecks and thus serving more requests per second (req/s) more quickly (Section VII-B).

Avoids Provider Lock-Ins: The designer's dilemma has prompted several providers of cloud services to develop and offer scalable "key-value" stores that run on top of distributed hash tables (DHTs). They offer transparent scalability at the expense of sacrificing the full functionality of established RDBMs (including the use of a powerful query language like SQL). However, systems like Amazon's SimpleDB and Google's BigTable require using APIs that are tied to a particular cloud provider and thus suffer from poor portability that can lead to architectural lock-ins to a particular cloud provider [9]. Cross platform key-value stores like Cassandra do not cause lock-in concerns, but face performance problems as we will argue shortly. SPAR is implemented as a middleware that is platform-agnostic and allows the developer to select its preferred datastore, e.g., Key-Value stores or RDBMS.

B. How Does SPAR Work?

In a nutshell, through *joint partitioning and replication*. By partitioning the data we make sure that the underlying community structure is preserved as much as possible, i.e., data of a user need to reside on the same server with the data of her neighbors. Most of the relevant data for a user in an OSN are one hop away (friends, followers, etc.). By replicating users that belong to multiple partitions and ensuring that data pertaining to all one-hop neighbors of a user are colocated with the user, we ensure local semantics at the data layer.

1) Simple Example: We use an example (Fig. 1) to communicate our main point that the two operations, partitioning and replication, need to be performed jointly. On the top of Fig. 1, we depict a social graph with 10 profiles and 15 edges (bidirectional friendship relationships). The graph includes two distinct communities connected through "bridge" nodes 3 and 4. For exposition, we assume all profiles have one unit of memory. To service one read to a profile, the profiles of "friends" are contacted. We depict the physical placement of profiles on two servers under the following schemes: (a) Full Replication (FR); (b) Random (or Hash based) Partitioning (RP); (c) Random Partitioning with Replication of missing neighbors (RPR); (d) our proposed Social Partitioning and Replication (SPAR). We summarize the memory and network cost (interserver traffic) of the different solutions assuming unit-sized profiles and a read rate of 1 for all profiles.

We make the following observations. RP minimizes the replication overhead (0 units) and thus can be implemented using the cheapest possible servers in terms of memory. On the downside, RP leads to the highest aggregate network traffic due to reads (10 units as servers need to communicate to service reads) and thus increases the network I/O cost of the servers and the networking equipment that interconnects them. Things get reversed with FR. In this case, network read traffic falls to 0 (as all data is local), but the memory requirements are maximized (10 units) as well as high write traffic for maintaining consistency. RPR too eliminates network read traffic, but still has high memory requirement and network write traffic. SPAR performs the best in terms of ensuring locality semantics with lowest overhead.

2) Summary of Results: The above toy example hints on the performance results of Section V based on workloads from Twitter, Orkut, and Facebook. We summarize them here.

- SPAR provides local semantics with the least amount of overhead (reduction of 200% in some cases over random) while reducing interserver read traffic to zero and balancing read and write operations across back-end servers.

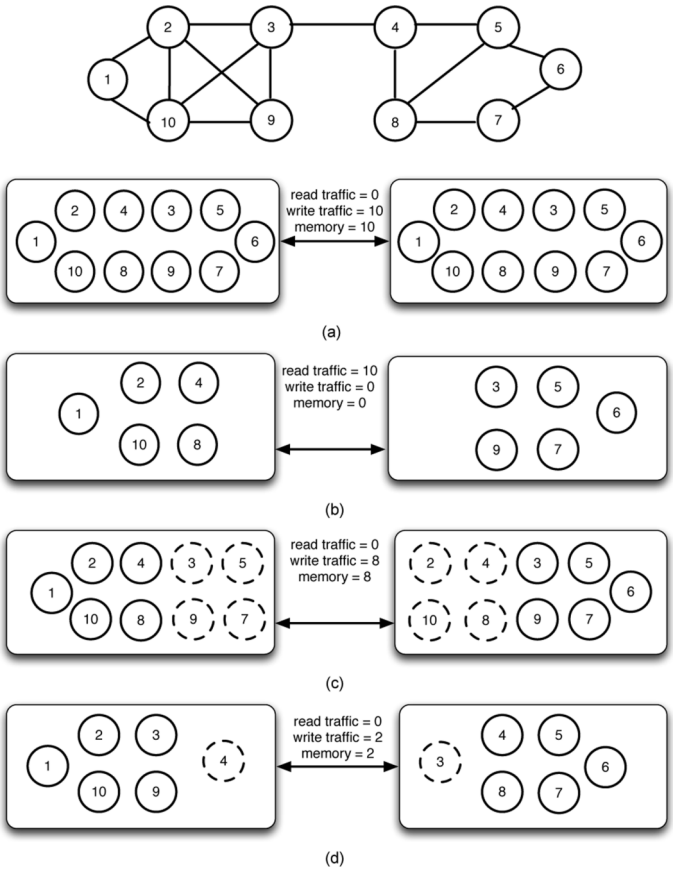


Fig. 1. (a) Full replication. (b) Partition using DHT (random partitioning). (c) Random partitioning (DHT) with replicating the neighbors, represented by hyphenated circles, helps decrease traffic. (d) SPAR with replicas represented as hyphenated circles.

- SPAR handles node and edge dynamics observed in OSNs with minimal overhead in the system. In addition, SPAR provides mechanisms for addition/removal of servers as well as handles failures gracefully, with low amount of overhead.
- In our implementation, SPAR serves 300% more req/s than Cassandra while reducing network traffic by a factor of 8. We also show substantial gains when we implement SPAR with MySQL.

III. Problem Statement

A. Requirements

The set of requirements that SPAR has to fulfill are the following.

Maintain Local Semantics: Relevant data for a user in most OSNs is her data in addition to the data related to her direct neighbors (e.g., followees' tweets, friends' status updates, etc.). To achieve local semantics, we need to ensure that for every *master replica* of a user, either a *master replica* or a *slave replica* of all her direct neighbors is colocated on the same server. The term *replica* is used to refer to a copy of the user's data. We differentiate between the *master* replica, serving read and write operations, and the *slave* replica, required for redundancy and to guarantee data locality.

Balance Loads: There are two distinct types of reads that occur in the system: The first set of reads is issued by users for their respective profiles. These reads are always handled by the *master* replica of the user. These reads, in turn, can lead to system reads on the profiles' of the user's neighbors to satisfy the initial read issued by the user. These system reads are handled by the master or slave replica of the user's neighbors colocated on the same machine. Write requests of a user are directed only to her *master* replica. Write operations need to be propagated to all her *slave* replicas for consistency. However, since masters handle much more load than slaves, we can obtain an approximately balanced load by distributing just the masters (hence balancing reads issued by users and writes) evenly among the set of back-end servers.

Be Resilient to Machine Failures: To cope with machine failures, we need to ensure that all masters have at least K slaves that can be used instead of the master at times of failure.

Be Amenable to Online Operations: OSN graphs are highly dynamic with new users joining constantly, followed by graph densification [26], fundamentally altering the underlying graph structure. Additionally, the infrastructure hosting the OSN may undergo changes like servers being added or existing ones upgraded. To handle such dynamics, a solution needs to be responsive to such dynamics (online) and *simple*, quickly converging to an efficient assignment of users to servers.

Be Stable: Given that we are operating in a dynamic setting, we need to ensure that a solution, in addition to being simple, is also stable. For example, addition of a few edges should not lead to a cascade of changes in the assignment of masters and slaves to servers.

Minimize Replication Overhead: We define *replication overhead* as the average number of replicas created per user. The overall performance of the system is highly related to the number of replicas in the system. For example, having more replicas increases the database size hosted at each server, resulting in slower execution of queries. Moreover, a higher number of replicas increases the amount of network traffic for keeping replicas up to date or conversely decreases their consistency [17]. The latter is an emerging concern given the large number of applications using OSNs that may need stricter consistency requirements than the OSNs were originally thought to require.

B. Formulation

Given the above requirements, we formulate an optimization problem to minimize the number of required replicas. We use the following notation. Let $G = (V, E)$ denote the social graph representing the OSN, with node set V representing user profiles and edge set E representing (friendship) relationships among profiles. Let $N = |V|$ denote the total number of profiles, and M the number of available (virtual) machines for hosting the OSN.

We cast the problem as an integer linear program, where p_{ij} denotes a binary decision variable that becomes 1 iff the primary of user i is assigned to partition j, $1 \leq j \leq M$. Also, r_{ij} denote a similar decision variable for a replica of user i assigned to partition j. Finally, let the constants $\epsilon_{ii'} = 1$ if $\{i, i'\} \in$

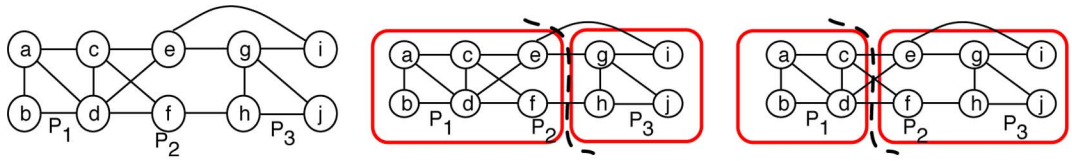


Fig. 2. Illustrative example on why minimizing edges between partitions does not minimize replicas. *(center)* A cut along P_2 and P_3 minimizes the edges, *(right)* while a cut along P_1 and P_2 minimizes the replicas.

E capture the friendship relationships. We can now state the Min_Replica problem as follows:

$$\min \quad \sum_i \sum_j r_{ij}$$

$$\text{s.t.} \quad \sum_{\forall j} p_{ij} = 1 \tag{1}$$

$$p_{ij} + \epsilon_{ii'} \leq p_{i'j} + r_{i'j} + 1 \qquad \forall i, j, i' \tag{2}$$

$$\sum_i (p_{ij}) = \sum_i (p_{i(j+1)}), \qquad 1 \leq j \leq M-1 \tag{3}$$

$$\sum_j r_{ij} \geq k \qquad \forall i \in V. \tag{4}$$

Constraint (1) in the above formulation ensures that there is exactly one master copy of a user in the system. Constraint (2) ensures that all neighbors (masters or slave of the neighbors) of a user are on the same machine. Constraint (3) tries to distribute equal number of masters across the machines, and Constraint (4) encodes the redundancy requirement.

Lemma 1: Min_Replica is NP-hard.

A simple reduction from the graph Min-bisection problem [16] can be used to show this. The proof is in the Appendix. The min-bisection problem has an $O(\log^2 N)$ approximation [15].

C. Why Graph/Social Partitioning Falls Short

An obvious set of candidates that can be used to address the problem we described in Section III-B includes graph-partitioning algorithms [10], [23] and algorithms that find communities in social graphs [12], [27], [31]. These algorithms either work by trying to find equal-sized partitions of a graph such that the interpartition edges are minimized [10], [23], or they rely on optimizing a metric called modularity that represents the quality of partitions produced [12], [31].

The main reasons why these methods are inadequate for our purpose are as follows.

- Most of these algorithms are offline [10], [12], [31] and cannot handle the dynamics of OSNs.
- Algorithms based on community detection are known to be extremely sensitive to input conditions, with nodes being assigned to different partitions/communities with little changes to the structure [24]. In other words, they do not produce *stable* solutions.
- The graph-partitioning algorithms mentioned above minimize interpartition edges. This, however, does not guarantee local semantics. One can of course add replicas *post hoc* producing partitions, guaranteeing local semantics, but as we will show in Section V, this leads to high replication overhead.
- Likewise, reducing the number of interpartition edges does not reduce the number of replicas. Consider, for instance, Fig. 2, where minimizing the number of edges will induce a cut along the partitions on the right side, while a cut along the left side is preferable to minimize replicas. As we will show later in Section V, minimizing interpartition edges indeed leads to poorer results. Motivated by the shortcomings of the existing solutions, we present our online solution in Section IV.

IV. SPAR: Joint Partitioning and Replication on the Fly

We propose a greedy heuristic as a solution to address the requirements described in Section III.

A. Overview

We assume that users represent nodes in a graph, and edges are formed when users create links with other users. The algorithm reacts to additions and removals of nodes, edges, and servers (six different events in total). Most of these events involve an *edge addition*. Edge addition event requires calculating the number of replicas and node movements needed to reconstruct the local semantics required by the two edge-related master nodes. A greedy choice is made to minimize replicas subject to the constraint of balancing the number of masters hosted on servers. A thorough complexity analysis of our greedy scheme would require competitive analysis, comparing against an optimal offline algorithm, which does not exist. This is beyond the scope of this paper. Instead, we focus on an empirical approach.

B. Detailed Description

The algorithm uses S_{ij}, the number of friends of node i in partition j, and R_{ij}, a binary value that becomes 1 iff i is replicated at j. In the average case, the required information is proportional to the product of the average node degree and the number of servers. The worst-case computational complexity of the algorithm is proportional to the highest node degree. The algorithm runs in the Partition Manager module as explained in Section VI-B. Once partitions are produced, we assign partitions to servers.

Node Addition: A new node is assigned to the partition with the fewest number of masters. In addition, K slaves are created and assigned to random partitions.

Node Removal: When a node is removed (that is, a user is deleted), its master and all its slaves are removed. The states of the nodes that had an edge with it are updated.

Edge Addition: When a new edge is created between nodes u and v, the algorithm checks whether both masters are already colocated with each other or with a master's slave. If so, no further action is required.

If not, the algorithm calculates the number of replicas that would be generated for each of the three possible configurations: 1) no movements of masters, which maintains the *status quo*; 2) the master of u goes to the partition containing the master of v; 3) the opposite.

Let us start with configuration 1. In this case, a replica is added if it does not already coexist in the partition of the master of the complementary node. This can increase replicas by 1 or 2, and can occur if nodes v or u already have relationships with other nodes in the same partition or if there exist extra slaves of v or u for redundancy. This is important since, as we will see later, the end result is that SPAR needs fewer overall replicas to ensure both local semantics and K redundancy.

In configuration 2, no slave replicas are created for u and v since their masters will be in the same partition. However, for the node that moves, in this case u, one might have to create a slave replica of itself in its old partition to service the master of the neighbors of u that were left behind in that partition. In addition, the masters of these neighbors will have to create a slave replica in the new partition—if they do not already have one—to preserve the local semantics of u. Finally, the algorithm removes the slave replicas that were in the old partition only to serve the master of u since they are no longer needed. The above rule is also subject to maintaining a minimum number of slave replicas due to redundancy K: The old partition slave will not be removed if the overall system ends up with less than K slaves for that particular node. Configuration 3 is just the complementary of 2.

The algorithm greedily chooses the configuration that yields the smallest aggregate number of replicas subjected to the constraint of load-balancing the master across the partitions. More specifically, configurations 2 and 3 also need to ensure that the movement either happens to a partition with fewer masters or to a partition for which the savings in terms of number of replicas of the best configuration to the second best one is greater than the current ratio of load imbalance between partitions.

Fig. 3 illustrates the steps just described with an example. The initial configuration (upper left subplot) contains six nodes in three partitions. The current number of replicated nodes (empty circles) is four. An edge between node 1 and 6 is created. Since there is no replica of 1 in M3 or replica of 6 in M1 if we maintain the *status quo*, two additional replicas will have to be created to maintain the *local semantics*.

The algorithm also evaluates the number of required replicas that are required for the other two possible configurations. If node 1 were to move to M3, three replicas would need to be created in M3 since only two out of the five neighbors of node 1 are already in M3. In addition, the movement would allow removing the slave of node 5 from M1 because it is no longer needed. Consequently, the movement would increase the total number of replicas by $3 - 1 = 2$, yielding a new total of six replicas, which is worse that maintaining the *status quo* of masters.

In the last step, the algorithm evaluates the number of replicas for the third allowed configuration: moving the master of node 6 in M1. In this case, the replica of node 5 in M3 can be removed

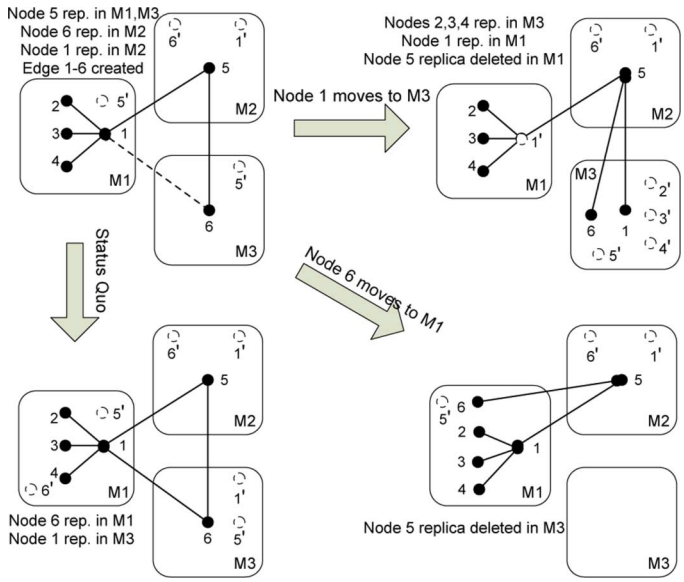


Fig. 3. SPAR online sketch.

because it already exists in M1 and no other node links to it in M3, thus no replica needs to be created. The change in the number of replicas is -1, yielding a total of three replicas.

Therefore, moving 6 to M1 minimizes the total number of replicas. However, such a configuration violates the load-balancing condition and cannot be performed. Thus, the final action is not to move (status quo) and create an additional two replicas.

Edge Removal: When an edge between u and v is removed, the algorithm consequently removes the replica of u in the partition holding the master of node v if no other node requires it, and vice versa. The algorithm checks whether there are more than K slave replicas before removing the node so that the desired redundancy level is maintained.

Server Addition: Unlike the previous cases, server addition and removal do not depend on the events on the social graph, but are externally triggered by system administrators or detected automatically by the system management tools.

There are two choices when adding a server: 1) force redistribution of the masters from the other servers to the new one so that all servers are balanced immediately; or 2) let the redistribution of the masters be the result of the of node and edge arrival processes and the load-balancing condition.

In the first case, the algorithm will select the $\frac{N}{M^2+M}$ least replicated masters from the M server and move them to the new server $M+1$. After the movement of the masters, the algorithm will ensure that for the masters that moved to the new server, there is a slave replica of their neighbors to *guarantee* the local data semantics. This mechanism guarantees that the master across all the $M+1$ are equally balanced. However, it may not provide a minimum replication overhead. Thus, in addition, for a fraction of the edges of the masters involved, the algorithm triggers a system-replay edge-creation event that reduces the replication overhead. As we will see later in Section VII, this provides good replication overhead even under server dynamic events.

In the second case, the algorithm does nothing else to increase the number of available servers. The edge/user arrival will take care of filling the new server with new users that in turn attract

old users when edges are formed to them. This leads to an eventual load balancing of the masters across servers without enforcing movement operations as long as the OSN continues to grow.

Server Removal: When a server is removed, whether intentionally or due to a failure, the algorithm reallocates the $\frac{N}{M}$ master nodes hosted on that server to the remaining $M-1$ servers equally. Note that in the case of a failure, the master replicas will have to be retrieved from servers hosting slave replicas as explained in Section VI-B.

The algorithm decides the server in which a replica would be promoted to master based on the ratio of its neighbors that already exist on that server. Thus, highly connected nodes, with potentially many replicas to be moved due to data local semantics, get to choose first to which server they go. The remaining ones are placed wherever they fit following simple water-filling strategy. As we will see in Section VII, this strategy ensures equal repartition of the failed masters while maintaining a small replication cost.

V. Measurement Driven Evaluation

A. Evaluation Methodology

1) Metrics: Our main quantitive evaluation metric is the *replication overhead* r_o for guaranteeing local semantics and at least K replicas for fault tolerance, as described in Section III. We report the average replication overhead over all users. We also consider the number of node movements, i.e., replica transmissions between machines, during the operation of SPAR online.

2) Datasets: We use three different datasets, each one serving a different purpose to evaluate individual performance metrics. *Twitter:* We collected a dataset consisting of a crawl of Twitter conducted between November 25–December 4, 2008. It comprises 2 408 534 nodes and 38 381 566 edges. The dataset contains approximately 60% of Twitter at the time of the crawl, but it does not contain precise information on edge-creation events. Note that the Twitter graph is directed. The dataset contains 12 million tweets generated by the 2.4 million users during the time of the collection. Although there exist larger datasets [25], they are topic-specific and do not contain all the tweets of individual users.

Facebook: We also used a public dataset of the New Orleans, LA, Facebook network [36]. It includes nodes, friendship links, as well as wall posts and was collected between December 2008 and January 2009. The data consist of 60 290 nodes and 1 545 686 edges. This dataset includes edge-creation timestamps between users as this information is captured as a wall post—however, this information is not available for all users. Therefore, we filtered the dataset and retained a subnetwork containing complete information for 59 297 nodes and 477 993 edges. This dataset permits us to study the dynamic evolution of the graph.

Orkut: This dataset consists of 3 072 441 nodes and 223 534 301 edges and was collected between October 3–November 11, 2006. More information about this dataset can be found in [29]. This is the largest of the three datasets.

3) Algorithms for Comparison: We compare SPAR against the following algorithms.

Random Partitioning: Key-Value-based stores like Cassandra, HBase, MongoDB, SimpleDB, etc., partition data randomly across servers. To the best of our knowledge, random partition is the default used in most commercial systems [22].

Graph Partitioning: There exist several offline algorithms for partitioning a graph into a fixed number of equal-sized partitions in such a way that the number of interpartition edges gets minimized [10], [23]. We use METIS [23], which is known to be very fast and yield high-quality partitions for large social graphs. [27].

Modularity Optimization (MO+) Algorithms: We also consider an offline community detection algorithm [12] built around the modularity metric [30], [31]. We modified it in order to be able to create a fixed number of equal-sized partitions [32]. Our modified version, called MO+, operates by grouping the communities in partitions sequentially until a given partition is full. In the case a community is larger than the predefined size, we apply MO recursively [12] to the community.

4) Computation of Results: The performance metrics are generated as follows.

- The input is formed by a graph (one of the datasets described earlier), the number of desired partitions M, and the desired minimum number of replicas per user's profile K.
- The partitions are produced by executing each one of the algorithms on the above-mentioned input.
- As we require local semantics, for the offline algorithms, we process the partitions obtained by all algorithms (except SPAR) and then add replicas when the master of a user is missing some of its neighbors in the same partition. Thus in effect, we are comparing against RPR, described in Section II-A.
- Since we only have edge-creation times for Facebook and we use those for SPAR. However, for both Twitter and Orkut, where we do not have that information, we use random permutations of the edges. We note here that we do not observe any qualitative differences when we use different permutations, even with Facebook.

B. Evaluation of Replication Overhead

Fig. 4 summarizes the replication overhead of the different algorithms over the different datasets for $K = 0$ and 2 and different numbers of servers, from 4 up to 512. We see that SPAR achieves much smaller replication overhead than all the other algorithms, including offline graph partitioning and community detection algorithms. The relative ranking of algorithms from best to worse is SPAR, MO+, METIS, and Random. Looking at the absolute value of the replication overhead, we see that it increases sublinearly with the number of servers. This means that adding more servers does not lead to a proportional increase in per-server resource requirement (as given by r_{o}). The sublinear increase of the overhead implies that system scales smoothly by just adding more servers. We evaluated SPAR and other schemes using *weighted* graphs, wherever the information was available. We used node degrees to assign weights to

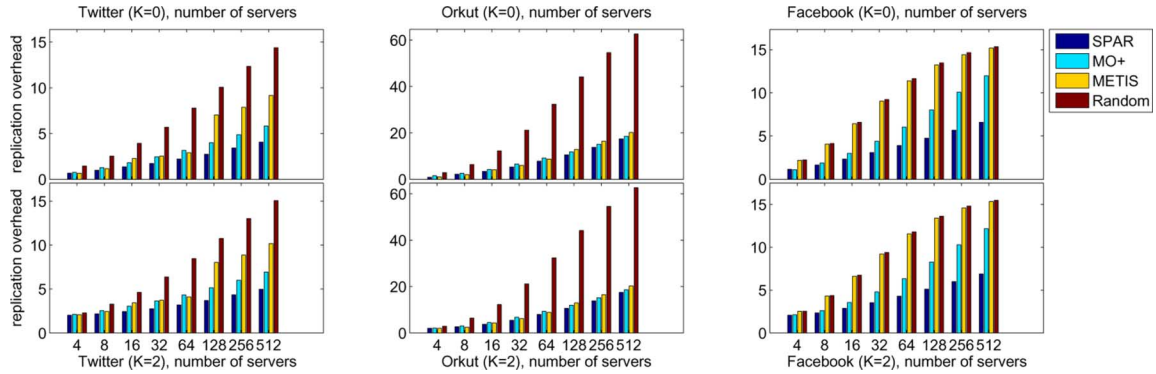


Fig. 4. Replication overhead (r_o) for SPAR, METIS, MO+, and Random for the empirical data of *(left)* Twitter, *(center)* Orkut, and *(right)* Facebook. The upper graphs show the results for $K = 0$, the lower graphs show the results for $K = 2$. K is the number of replicas for redundancy.

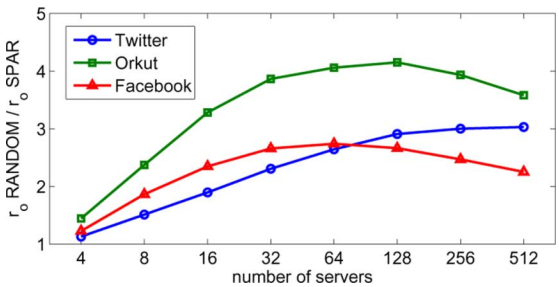


Fig. 5. SPAR versus random for $K = 2$.

nodes, accounting for profile sizes. We note that the results were similar.

SPAR naturally achieves a lower replication overhead since this is the optimization objective, whereas the competing algorithms optimize for interserver edges and thus end up needing more replicas for guaranteeing local semantics in each partition in the post-processing step.

Low-Cost Redundancy: An important property of SPAR is that *it achieves local semantics at a discounted cost by reusing replicas that are needed for redundancy anyway, or the other way around*. To see this, let us focus on the case of 32 servers. If no fault tolerance is guaranteed ($K = 0$), then the replication overhead of SPAR to ensure local semantics is 1.72. If we require its profile to be replicated at least $K = 2$ times, then the new replication overhead is 2.76 (rather than 3.72). Out of the 2.76 copies, 2 are inevitable due to the redundancy requirement. Thus, the local semantics were achieved at a lower cost $2.76 - 2 = 0.76$ instead of 1.72. The gain comes from leveraging the redundant replicas to achieve locality, something only SPAR does explicitly.

We now focus our comparison on Random versus SPAR since Random seems to be used by many key-value database systems. In Fig. 5, we depict the ratio between the overhead of Random over that of SPAR. In the case of the Twitter dataset, we see improvements varying from 12% for a partition of four servers (it is low because we have $K = 2$, which means three replicas of each node in four machines) to 200% in the case of 512 servers. For Orkut and Facebook, the improvement ranges from 22% and 44% to 174% and 315%, respectively.

Such smaller overheads permit SPAR to achieve a much higher throughput, as we will demonstrate in Section VII by running a real implementation of SPAR against Cassandra (Facebook's database) and MySQL.

One may wonder about SPAR's performance with highly skewed social graphs, graphs where some users have a large number of connections. This is not a concern as some OSNs like Facebook impose an upper limit on the number of contacts (5000) a user can have, thereby limiting the skewness. For Twitter, there are users (like `@oprah`) who have millions of followers, but these users themselves follow way fewer users. Therefore, due to the directionality of the graph, a superuser like `@oprah` will be replicated across multiple servers, but to maintain locality for `@oprah`, few replicas need to be created.

C. Dynamic Operations and SPAR

We have so far shown that SPAR online outperforms existing solutions in terms of replication overhead. We now turn to other system related requirements stated in Section III-A.

1) Delicate Art of Balancing Loads: We focus on how replicas are distributed across users. Let us take the example of Twitter with 128 servers and $K = 2$. The average replication overhead is 3.69: 75.8% of the users have 3 replicas, 90% of the users have 7 or less replicas, and 99% of the users have 31 or less replicas. Out of the 2.4 million users, only 139 need to be replicated across the 128 servers. Next, we look at the impact of such replication distribution on read and write operations.

Reads and Writes: Reads are only conducted on masters. Therefore, we want to see whether the aggregate read load on servers due to reads are balanced or not. The above load depends on the distribution of masters among servers and on the read patterns of the users. SPAR online yields partitions where the coefficient of variation (COV) in terms of masters is 0.0019, showing that our online heuristic is successful at balancing masters across servers. We note here that such systems have a high read-to-write requests ratio. Hence, we focus on balancing for reads in SPAR. However, we also test against workload imbalances and examine write patterns. The COV of writes per server is 0.37, which indicates that they are fairly balanced across all servers and no single server serves a significant proportion of writes. We expect reads to be even more balanced across users as systems like Twitter handle reads that are generated automatically through API calls via periodic polling (90% of Twitter traffic is generated via its API). Also,

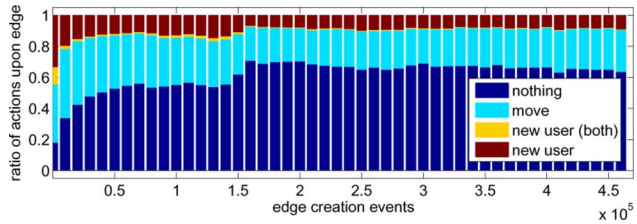


Fig. 6. Ratio of actions peformed by SPAR as edge-creation events occur for Facebook.

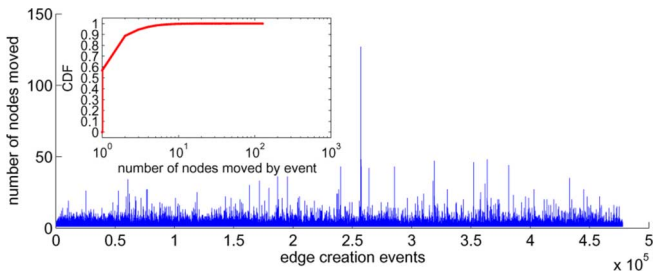


Fig. 7. Number of movements per edge-creation event for Facebook.

there is low correlation ($\rho \approx 0.26$) between heavy writers and their number of slave replicas, thus they do not present a big problem for the system.

2) Moving Nodes Around: Next, we turn our attention to the footprint of SPAR in terms of user's profile transmissions.

In Fig. 6, we show a stacked-bin time series of action taken by SPAR online upon each edge arrival event for the Facebook dataset with $K = 2$ and 16 servers. We see that following a transient phase during which the network builds up, SPAR online enters a steady-state phase in which 60% of edge arrivals do not create any transmissions. In the remaining 40% of the cases, data get transmitted.

In Fig. 7, we plot the number of transmissions per edge arrival, with its cumulative distribution function (CDF) as an inset. We can see that whenever a transmission occurs, an overwhelming majority (90%) of the movement results in very few transmissions, one or two. The maximum we observed was transmitting the data of 130 users.

Fig. 8 summarizes the transmission costs on the system for Facebook, Orkut, and Twitter from 4 servers to 512. The left plot in Fig. 8 depicts the average fraction of *do nothing* actions that involves no transmission of data discounted the transient phase. The right plot in Fig. 8 depicts the total of number of movements divided by the number of edges. These figures show that for each configuration of number of servers and datasets, the footprint remains low.

3) Adding/Removing Servers:

Adding Servers: When a server is added, we can use one of two policies: wait for new arrivals to fill up the server, or redistribute existing masters from other servers into the new server. We will study the overhead of such policies in terms of replication costs.

We start with the first case, where we go from 16 servers to 32 by adding one server every 150 000 new users. This strategy yields a marginal increase of r_o (2.78) against the r_o (2.74)

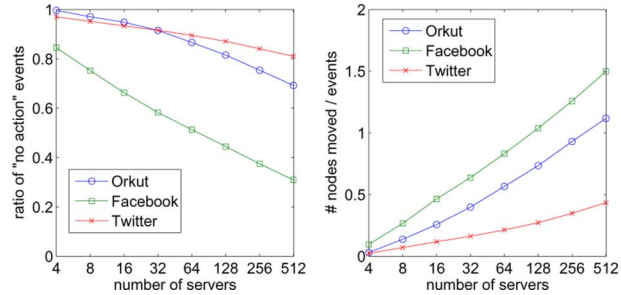


Fig. 8. Movement costs for various datasets.

that would have been obtained if the system had been dimensioned for 32 servers from the start. This shows that SPAR is able to achieve an efficient state independently on how servers are added. Although it does not do an explicit redistribution, the COV of the number of masters per server remains very low at the end on the trace (0.004). This shows that we can gracefully add new servers with no overhead.

In the second experiment, we tested an extensive upgrade in the infrastructure. We double the number of servers and enforce redistribution. This causes a reduction of the number of masters per server by half while being load-balanced. We tested the addition of 16 servers at the end of the trace and after 50% of the trace replayed. Doubling the number of servers leads to an expected transmission of half the masters. SPAR, however, needs to transmit additional slaves to maintain local semantics. Adding a server also causes an increase on the final replication overhead. For instance, doubling the initial 16 servers at 50% of the trace produces a transitory increase of 10% on the replication overhead that is progressively reduced as new edges are added, reaching 2.82, only a 2% more overhead than that caused with all 32 servers from the start. With proactive reshuffling of edges as described in Section IV, the added overhead caused by adding machines becomes almost insignificant.

Removing Servers: Next, we test what happens when a machine is removed. The average number of transmissions is 485 000 with a marginal increase on the r_o from 2.74 to 2.87. We can further reduce r_o to 2.77 at the cost of additional an 180 000 transmissions if we replay the edges of the nodes affected by the machine removal. One might argue that server removal seldom happens as systems are usually not scaled down. The case of server removals due to failures is discussed in Section VI-B.

D. Placement of Partitions in the Cloud

So far, we have assumed that the OSN owns and controls the underlying physical infrastructure: the servers and the network connecting them. However, the emergence of cloud systems like Amazon EC2 has made outsourcing the hosting to the cloud a convenient and cheaper option, at the expense of ceding control. This loss of control can lead to unexpected performance. However, given some information about the cloud environment, we can perform the placement of partitions in such a way to minimize the impact of operating in an unpredictible environment.

In this section, we discuss the problem of placing the obtained set of partitions $\mathcal{P}$ on a set of virtual machines $\mathcal{M}$ contracted from a cloud infrastructure for hosting the system, $|\mathcal{M}| \geq |\mathcal{P}|$.

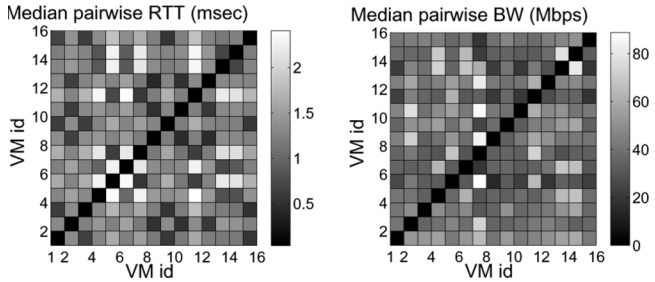


Fig. 9. Pairwise RTT and available bandwidth between 16 Amazon EC2 VMs instantiated in four availability zones. Results obtained using three `ping`'s and three `pathChirp`'s for each pair once per hour across a day.

Let $T_{jj'}$ denote the aggregate traffic going between partition j and j' due to updates from writes on primary replicas hosted at j. Let $d_{jj'}$ denote the communication cost between virtual machine j and j' over the datacenter network. Then, an optimal placement of partitions to virtual machines can be obtained by embedding the set of nodes $\mathcal{P}$ with a minimum distortion of cost (distance) with edge cost $1/T_{jj'}$ for edge (j, j') to a subset of $\mathcal{M}$ with edge cost $d_{jj'}$ for edge (j, j'). The traffic matrix T that defines the edge costs of the initial graph has no connection to physical distance, and thus the distance function obtained from $1/T_{jj'}$'s is nonmetric in the general case. Thus, we cannot use any of the known results on bounded distortion embeddings that exist for metric distance, e.g., [14], and consequently, we have to resort to heuristics that work well in practice.

We have experimented with several heuristics based on random, greedy, and local search algorithms, using as input $T_{jj'}$'s obtained by processing the datasets and the partitioning algorithms described earlier and $d'_{jj}s$ obtained by measurements between VMs purchased from Amazon EC2. We observed that, in practice, all of the above heuristics perform approximately the same. The reason behind this is the uniformity of our measured $d'_{jj}s$. In Fig. 9, we demonstrate this uniformity based on $d'_{jj'}s$ that correspond to round-trip delays (left) or bottleneck bandwidths (right) between 16 VMs in Amazon EC2 that we measured using `ping` and `pathChirp`. We requested four VMs in each one of four different availability zones offered by EC2 in the East US region. Within the same availability zone, the RTTs were below 0.5 ms, whereas across availability zones they were typically below 2 ms, both of which are insignificant compared to the total end-to-end delay to the end-user. Similarly, the bandwidth that we measured is several orders of magnitude greater (Fig. 9) than the maximum interpartition write traffic.

Overall, we have demonstrated that SPAR is able to distribute load efficiently and cope with both social network graph dynamics as well as physical (or virtual) machine dynamics at low overhead. We next describe the SPAR system architecture.

VI. SPAR System Architecture

We describe the basic architecture and operation of SPAR without detailing every aspect of the system. Fig. 10 depicts how SPAR integrates into a typical three-tier Web architecture. The only interaction between the application and SPAR is through the Middleware Library (*ML*). The *ML* needs to be called by

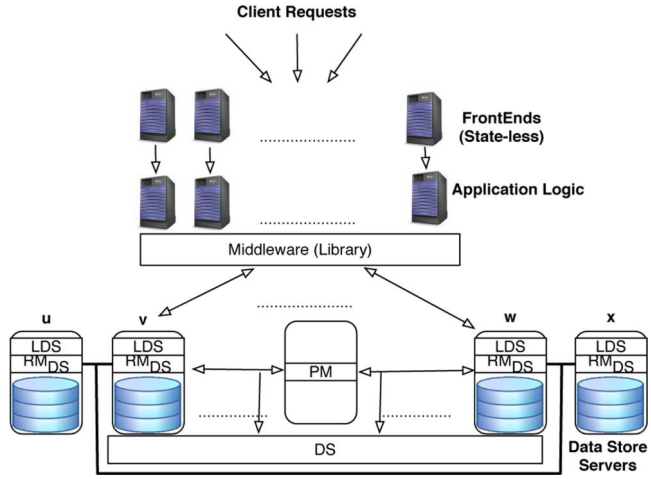


Fig. 10. Sketch of SPAR's architecture.

the application to know the address of the back-end server that contains the data to be read or written. Once the application has obtained the address, it uses the common interface of any datastore—e.g., a MySQL driver, Cassandra's API, etc.—and does not need to deal with how the data back end operates or scales. Making those operations scalable and transparent is the task of the remaining components of SPAR: the Directory Service *DS*, the Local Directory Service *LDS*, the Partition Manager *PM*, and the Replication Manager *RM*.

A. System Operations

Data Distribution: The data distribution is handled by the *DS*, which returns the server that hosts the master replica of a user through a key table lookup: $H_{\mathrm{m}}(\mathrm{key}) \rightarrow u$. Additionally, the *DS* also resolves the list of all servers where the user has replicas in $H_{\mathrm{s}}(i) \rightarrow \{u, v, \ldots, w\}$.

The Directory Service is implemented as a DHT with consistent caching and is distributed across all servers in the data back end. The Local Directory Service contains a partial view of the *DS*. Specifically, the $\frac{N(1+r_{\mathrm{o}})}{M}$ of the lookup table H_{m} and $\frac{N}{M}$ of lookup table H_{s}. The *LDS* only acts as cache of the *DS* for performance reasons and is invalidated by the *DS* whenever the location of a replica is updated.

Data Partitioning: The *PM* runs the SPAR algorithm described in Section IV and performs the following functions: 1) map the user's key to its replicas, whether master or slaves; 2) schedule the transmission of replicas; and 3) redistribute replicas in the case of a server addition or removal.

PM maintains a lookup table H_S containing the set S_{ij} described in Section IV. This corresponds to $4 + (2 + 4)(1 + r_{\mathrm{o}})$ bytes per user, or 30 MB per million users in the case of Twitter and 128 servers. SPAR's algorithm would allow the *PM* to be distributed. However, for simplicity, we have decided to implement a centralized version of *PM* and run multiple mirrors that act upon failures of the main *PM* to avoid single point of failure. *PM* is the only component that can update the *DS*. Note that this simplifies the requirements to guarantee global consistency on the *DS* because only the main *PM* can write to it. There can be multiple parallel instances of *PM* running simultaneously to handle the load.

Data Transmissions: Data transmission takes place when a replica needs to be moved or copied from one server to another. After a movement, all replicas undergo reconciliation to avoid inconsistencies that could have arisen during the movement, i.e., the user writing data while its master changes its location. The same applies when a replica is scheduled to be removed. We do not propose a new mechanism to handle such reconciliation events, but instead rely on the semantic reconciliation that uses versioning proposed in other distributed systems (e.g., Dynamo [13]).

Data Replication and Consistency: The Replication Manager *RM* runs on each server of the data back end. The main responsibility of *RM* is to propagate the writes that take place on a user's master to all her slaves using an eventual consistency model, which guarantees that all the replicas will—with time—be in sync. SPAR works within the confines of the CAP theorem, trading off consistency for availability and partition tolerance (by relying on replicas).

Note that since SPAR relies on a single-master and multiple-slaves configuration, we avoid the inconsistencies derived from maintaining multiple masters. In addition, this also has the benefit that inconsistencies can only arise due to failures or due to movements produced by edge, node, or server arrivals and not as part of the regular operations of read and writes.

RM is not replacing the datastore, but different datastores need different implementations of the *RM* to adapt to its interface. In this first iteration of SPAR, we provide the implementation of *RM* for MySQL and Cassandra although other datastores can be easily supported (i.e., Postgres, Memcache, etc.).

Adding and Removing Servers: The *PM* also controls the addition and/or removal of servers into the back-end cluster on demand. This process is described in Section IV and evaluated in Section V.

Handling Failures: Failures in the servers running the data back end will happen, either because of a server specific failure, e.g., disk, PSU failure, etc., or a failure at the level of the datacenter, e.g., power outages, network problems, etc.

SPAR relies on a *heartbeat*-like system to monitor the health of the data back-end servers. When a failure is detected, the *PM* decides the course of action based on the failure management policy set by the administrator. We consider two types of policies, one that reacts to *transient* failure and another one to *permanent* failures.

Permanent failure events are treated as server removals. In this case, slave replicas of the master that went down are promoted to masters, and all their neighbors are recreated. For more details, see Sections IV and V.

In the case of *transient failures* that are short-lived, one potential option is to promote slaves of the masters that went down without recreating their neighbors, therefore temporarily sacrificing *local data semantics*. In this case, the system suffers a graceful degradation since we can leverage a nice property of SPAR. Such property entails that the server hosting a replica of one of the failed masters will also contain a large portion of the total master's replicas. Therefore, while the promotion of a slave does not guarantee local semantics, it does provide a user with access to most of her neighborhood. For instance, for the Twitter dataset, 65% of users have 90% or more of their one-hop neighbors present on the server that hosts the best possible replica. This solution, however, should only be applied for cases of extremely short-lived outages, otherwise the user experience of the OSN would suffer, and the system administrator is better off implementing the permanent failure scenario.

Current SPAR replication is not meant for *high availability*, but for redundancy and local data semantics. However, high availability in SPAR could be obtained in two ways. One is to modify the formulation of the problem so that there are at least K' replicas where the *data local semantics* is maintained while minimizing Min_Replica. This is left for future work. The other option is a brute force approach that mirrors the SPAR system under $k = 0$ as many times as desired. For instance, to obtain three redundant copies with full local semantics in the case of Twitter, this translates into an r_{o} increase from 3.20 to 6.63 for 128 servers.

B. Implementation Details

The *RM* sits on top of a datastore and controls and modifies the queries. The read events (selects) are forwarded directly to the datastore without delay. The write events (updates, inserts, deletes) are analyzed and can be altered both for replication and performance reasons. To illustrate the inner workings, we use an example of an OSN application and a write operation, using MySQL as an example datastore.

A user i wants to create a new event w, which generates the following command to be inserted in MySQL: *insert into event* (i_{id}, w_{id}, w), where i_{id} and w_{id} are the pointers to the user and the event's content. The *RM* will react to this command by obtaining the target table *event*. The *RM* knows, through simple configuration rules, that the *event* table is partitioned, and thus the *insert* needs to be replicated to all the servers hosting the slave replicas of user i_{id}. The *RM* queries its *LDS* to obtain the list of servers to propagate the insert, and issues the same *insert* command to the local MySQL. Additionally, this event will be broadcast to all the neighbors of the user. For each contact j of i, the application will generate an *insert into event-inbox* (i_{id}, j_{id}, w_{id}), and the RM_{mysql} will replicate the same commands to all the appropriate servers.

VII. SPAR in the Wild: Evaluation

In this section, we study the performance of SPAR with a concrete implementation over two datastores: a traditional RDBMs system, MySQL, and a prime example of a key-value store, Cassandra. More specifically, we compare SPAR against random partitioning and full replication in our implementation. As a reference OSN application, we use Statusnet, an open-source Twitter implementation that relies on a centralized architecture (PHP and MySQL/Postgres).

Our testbed is a cluster of 16 low-end commodity machines—"little engine(s)," interconnected by a Giga-Ethernet switch. Each machine has a Pentium Duo CPU at 2.33 GHz with 2 GB of RAM and a single hard drive. We load the Twitter data described in Section V-A.2 in the datastores.

A. Evaluation With Cassandra

Statusnet is designed to run on RDBMs as the back-end datastore. Therefore, to evaluate SPAR with Cassandra, we need to

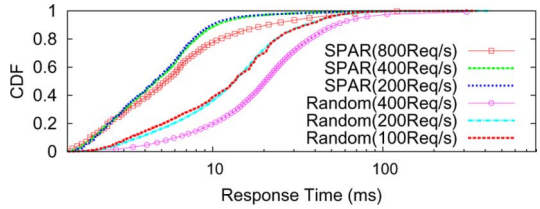


Fig. 11. Response time of SPAR instantiation in Cassandra.

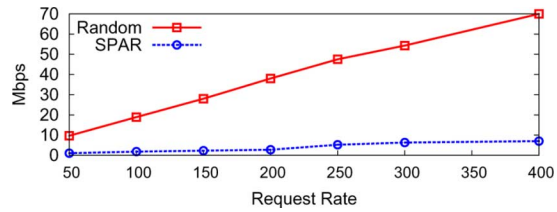


Fig. 12. Network activity in SPAR instantiation in Cassandra.

reproduce the functionality on Statusnet for the data-model specific of Cassandra (ver. 0.5.0). To emulate Statusnet functionality, we define a data scheme that contains information about users, updates (;tweets), and the list of update streams to which the users subscribe. We implement the data scheme using different *columns* and *super columns*.

To implement SPAR, first we disable the default random partitioning algorithm of Cassandra by creating independent instances of Cassandra. The Cassandra nodes in our system do not communicate with each other, so we have full control of the location of the information (users and updates). To implement the Directory Service, we use an extra instance of Cassandra as our key-value store, showing that we can piggyback on the underlying datastore infrastructure. The *DS* is distributed across all servers to avoid bottlenecks. A *PM* node runs on an independent server handling new edges and nodes and eventual migrations.

We wrote the SPAR middleware that connects Cassandra through a Thrift interface. The middleware provides basic operations needed for the normal operations of Statusnet. For instance, the canonical operation for Statusnet, retrieving the last 20 updates for a given user, is handled by the middleware using these three steps: 1) randomly select a Directory Service node and request the location of the master of the user by using the *get* primitive of Cassandra; 2) connect to the node that hosts the master and perform a *get-slice* operation to request the *update-id*'s of the list of last 20 updates; and finally 3) do a *multiget* to retrieve the content of all the updates and return to Statusnet.

As noted earlier in our evaluation, we compare the performance of Statusnet with the SPAR instantiation on Cassandra and the standard Cassandra with random partition. We are interested in answering two questions: 1) What impact does SPAR have on response times as compared to random partitioning? 2) How much does SPAR reduce network traffic?

To answer these two questions, we perform the following set of experiments. We randomly select 40 000 users out of the Twitter dataset (that are loaded in Cassandra) and issue requests to retrieve the last 20 updates at a rate of 100, 200, 400, and 800 requests per second.

Examining Response Times: Fig. 11 shows the response time of SPAR and the default Cassandra using random partitioning. We can see that SPAR reduces the average response time by 77% (400 req/s). Additionally, we also note that SPAR can support a 99-percentile response time below 100 ms. SPAR provides the same quality of service for a request rate of 800 req/s, while Cassandra with the default random scheme can only provide such performance for 1/4 of the request rate.

Why does SPAR outperform random partitioning? One reason for the performance improvement is that Cassandra is limited by the worst server's delay. This could be due to the heavy interserver traffic for handling remote reads. However, our implementation runs on Gigabit Ethernet switches with no other background traffic, and hence network bandwidth is not a bottleneck. The commodity servers, however, often hit network I/O and CPU bottlenecks in trying to sustain high rates of remote reads. With SPAR, as relevant data is local, remote reads are not needed. A second and less obvious reason for the higher performance of SPAR has to do with its improved memory hit ratio that comes as a by-product of the nonrandom partitioning of masters. Indeed, a read for a user brings in memory from the disk of the data of the user as well as those of her friends. Given that masters of her friends are on the same server with a high probability, and that reads are directed to the masters, there is a good chance that a read for one of these friends will find most of the required data already in memory because of previous reads. The random partitioning scheme of Cassandra destroys such correlations and thus suffers from a lower memory hit ratio and slower disk access times.

Analyzing the Network Load: Fig. 12 depicts the aggregate network activity of the cluster under various requests rates. For random partitioning, update requests are spread among multiple nodes, and *multiget* operations (step 3) significantly increase the network load. In fact, SPAR provides a reduction of approximately 8 times of the network traffic (400 req/sec) compared to the default Cassandra implementation.

B. Evaluation With MySQL

We now turn our attention to MySQL, a traditional RDBMs system. The first question we probe is "Can SPAR scale an OSN application using MySQL?" This is important as it allows developers to continue using the familiar RDBMs framework without worrying about scaling issues. Specifically, we want to answer if we can make Statusnet deal with the demand of Twitter as of December 2008. We deliberately choose commodity servers to underscore the point that one can use SPAR with such commodity machines to reach Twitter-level demand rates.

We use MySQL ver. 5.5 together with the SQL data scheme used by Statusnet [5]. The schema contains SQL tables related to the users (table *user* and *profile*), the social graph (*subscription*), updates (*notice*), and the list of updates per user (*notice_inbox*). We adapted our Twitter dataset to the Statusnet data scheme so that it contains all information about users and updates. We retrieve the last 20 updates per user by performing a single query using a join on the *notice* and *notice_inbox* tables.

To stress-test our setup, we use *tsung* [6] and two machines that we use to emulate the activity for tens of thousands of concurrent users. We generate both read operations (retrieve the last

20 updates) and write operations (generate a new update and update the inboxes). Our experimental evaluation consists of multiple 4-min sessions where we query for the last updates of a random subset of users with a constant request rate. We make sure that every user is queried only once per session and that the requests are spread evenly among servers.

Comparison to Full Replication: First, we check whether a scheme that is based on *full* replication can work in practice. This would mean loading the entire Twitter dataset on all machines and measuring the number of users that the system can serve. The average 95th percentile of the response time per user is 113 ms for 16 users/s (one user per second per machine), 151 ms for 160 users/s, and 245 ms for 320 users/s. The 99th percentiles are even higher with 152 ms for 16 users/s. On the other hand, when we use SPAR, the cluster can serve more than 2500 users/s with a 99th percentile of less than 150 ms. This shows that SPAR using a MySQL datastore is able to withstand Twitter-scale read loads with a small cluster of commodity machines, while a full replication system cannot cope.

Adding Writes: To further stress-test the system, we introduce insertions of updates. We evaluate the effect of the insertion of 16 updates/s (1 update/s per machine). In this part, we evaluate only SPAR (with MySQL), as full replication using MySQL performs very poorly.

Note, that the insertion of a new update to the system can generate thousands of updates since the system needs to insert to the *notice_inbox* table one entry for every user that should receive the update. How we treat these inserts is crucial for the overall performance. A naive implementation that performs single or multiple inserts into the *notice_inbox* can completely saturate the system.

We group the inserts and control the rate at which we introduce them in the system. Under this scenario, we show that we can achieve a 95th-percentile response time below 260 ms for 50 users/s and below 380 ms for 200 users/s. We should note here that the median response time in both cases is very low, around 2 ms. Performance will only get better as we add more machines.

VIII. Related Work

Scaling Out: Scaling out in software is provided by current cloud providers like Amazon EC2 and Google's AppEngine, as well as companies like RightScale, by giving the ability to launch virtual instances as and, when needed, to cope with demand [4]. However, they provide scaling out for the *front end* of the application and the *back end* as long as the data in the back end is independent—we deal with scaling of the application *back end* when data is *not* independent by providing means to ensure local semantics at data level.

Key-Value Stores: Many popular OSNs today rely on Key-Value stores that are DHT-based to deal with scaling problems in the back end (e.g., Facebook uses Cassandra). While these key-value stores have addressed the problem of scalability, these datastores rely on random partitioning of back-end data, which can lead to poor performance in the case of OSN workloads (as we show in this paper). SPAR improves performance multifold over DHT-based stores as SPAR ensures relevant data is local while minimizing network traffic. Keeping data local also helps prevent issues like "multiget" holes observed in some systems [1].

Distributed File Systems and Databases: Distributing data for performance, availability, and resilience reasons has been widely studied in the file system and database systems community. Ficus [18] and Coda [33] are DFS systems that replicate files for high availability. Farsite is a DFS that achieves high availability and scalability using replication [8]. Distributed relational DB systems like Bayou [35] allow for disconnected operations and provide eventual data consistency. SPAR is a middleware-based solution that provides scalability and increases performance of back-end databases in the presence of highly interconnected data by exploiting the underlying structure present in OSNs. SPAR also provides eventual consistency [13], [35] while providing redundancy.

IX. Conclusion

Scaling OSNs is a hard problem because the data of users is highly interconnected and hence does not afford clean partitions. We propose SPAR, which is based on partitioning of the OSN graph combined with replication of users with the aim of ensuring local data semantics for all users. By local semantics, we mean that all relevant data of the direct neighbors of a user are colocated on the same machine as the user. This enables all queries for a user to be resolved locally on that machine.

Preserving local semantics has many benefits. First of all, it enables transparent scaling of the OSN at a low cost. Second, the performance benefits in requests per second served increase multifold as all relevant data is local, and sending queries over the network is avoided. Third, network traffic is reduced. We design and validate SPAR using real datasets from three OSNs and show that it performs well in terms of reducing the overhead and dealing with high dynamics experienced by an OSN gracefully. We implement a Twitter-like application and evaluate SPAR with MySQL and Cassandra using real datasets and show significant gains in terms of requests per second and reduction in network traffic.

Appendix
NP Hardness of Min_Replica

Assume we have an oracle A that provides a solution for the decision version of MIN_REPLICA in polynomial time. We need a problem B that is NP-complete and to show $B \leq_p A$. We perform the reduction from the graph bisection problem [16], where given a graph we need to bisect the set of vertices into two disjoint equal-sized sets/partitions such that the cost function between the sets (number of edges between the sets) is minimized. Define the generalized version of the bisection (Gen-Bis) problem to be where we are given a graph and a subset of "colored" vertices, bisect the entire set of vertices so that each set has an equal number of "colored" vertices and the number of edges between the sets is minimized. Clearly, the bisection problem reduces to Gen-Bis by coloring all vertices. Thus, Gen-Bis is NP-complete.

Consider the problem Replica-Bis, where we are given a graph and a set of colored vertices, and we need to bisect such that each set has the same number of colored vertices and the number of "bridge" vertices (bridge vertices are vertices

in either set with edges to the other set) are minimized. Our problem MIN_REPLICA is Replica-Bis for $M = 2$.

The reduction of Gen-Bis to Replica-Bis is as follows. We are given the graph for Gen-BIS, $G = (V, E)$, s.t. $N = |V|$. Without loss of generality, we assume that this graph is a regular graph, i.e., all vertices have the same degree d. The same degree corresponds to the minimum number of replicas per node, k. For each vertex in the graph of Gen-BIS, we create d copies. For every edge in E, say (u, v), we create an edge between a pair of the corresponding copy nodes (u_i, v_j), where $u_i \in (u_1, \ldots, u_d)$ and $v_j \in (v_1, \ldots, v_d)$. We have exactly E edges in this new graph.

Then, for each node in graph G, we create a clique of n^2 vertices. For each of the n^2 vertices in this clique, we add edges to all the d copies of the node to which the clique corresponds. In other words, the degree of each node in the clique increases to $n^2 - 1 + d$.

We feed this graph as input to the Replica-BIS problem with the copies as the colored vertices. Now observe that in any partition no two copies of the same vertex will be on opposite sides because straightaway we will create a "bridge" vertex that increases the number of boundary nodes to be $>= n^2/2$. With this restriction, we observe that the number of "bridge" vertices is exactly equal to twice the number of crossing edges in the original graph G. Hence, if we can solve Replica-BIS, we can solve Gen-BIS, so Replica-BIS must be NP-complete. Therefore, the decision version of MIN_REPLICA is NP-complete.

□

ACKNOWLEDGMENT

The authors thank the anonymous reviewers of ACM SIGCOMM 2010. They are very grateful to Z. Vagena, R. Sundaram, and Y. Zhang for providing valuable feedback. They also thank A. Mislove for making the Facebook data available.

REFERENCES

[1] High Scalability, "Facebook's memcached multiget hole: More machines != More capacity," 2009 [Online]. Available: http://highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more-machines-more-capacit.html

[2] High Scalability, "Friendster lost lead due to failure to scale," 2007 [Online]. Available: http://highscalability.com/blog/2007/11/13/friendster-lost-lead-because-of-a-failure-to-scale.html

[3] V. Agunati, "Notes From Scaling MySQL—Up or Out," 2008 [Online]. Available: http://venublog.com/2008/04/16/notes-from-scaling-mysql-up-or-out/

[4] RightScale, "RightScale," 2010 [Online]. Available: http://www.rightscale.com

[5] StatusNet, "Status Net," 2010 [Online]. Available: http://status.net

[6] N. Niclausse, "Tsung: Distributed load testing tool," 2010 [Online]. Available: http://tsung.erlang-projects.org/

[7] High Scalability, "Scaling Twitter: Making Twitter 10000 percent faster," 2009 [Online]. Available: http://highscalability.com/scaling-twitter-making-twitter-10000-percent-faster

[8] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, Jon, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," in *Proc. OSDI*, 2002, pp. 1–14.

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A Berkeley view of cloud computing," Tech. Rep. UCB/EECS-2009-28, 2009.

[10] S. Arora, S. Rao, and U. Vazirani, "Expander flows, geometric embeddings and graph partitioning," *J. ACM*, vol. 56, no. 2, pp. 1–37, 2009.

[11] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida, "Characterizing user behavior in online social networks," in *Proc. ACM SIGCOMM IMC*, 2009, pp. 49–62.

[12] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *J. Statist. Mech.*, vol. 10, p. P10008, 2008.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.

[14] J. Fakcharoenphol, S. Rao, and K. Talwar, "Approximating metrics by tree metrics," *SIGACT News*, vol. 35, no. 2, pp. 60–70, 2004.

[15] U. Feige and R. Krauthgamer, "A polylogarithmic approximation of the minimum bisection," *SIAM J. Comput.*, vol. 31, no. 4, pp. 1090–1118, Apr. 2002.

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: Freeman, 1979.

[17] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proc. ACM SIGMOD*, 1996, pp. 173–182.

[18] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page Jr., G. J. Popek, and D. Rothmeier, "Implementation of the Ficus replicated file system," in *Proc. USENIX Conf.*, 1990, pp. 63–72.

[19] J. Hamilton, "Geo-replication at Facebook," 2008 [Online]. Available: http://perspectives.mvdirona.com/2008/08/21/GeoReplicationAtFacebook.aspx

[20] J. Hamilton, "Scaling LinkedIn," 2008 [Online]. Available: http://perspectives.mvdirona.com/2008/06/08/ScalingLinkedIn.aspx

[21] High Scalability, "Why are Facebook, Digg and Twitter so hard to scale?," 2009 [Online]. Available: http://highscalability.com/blog/2009/10/13/why-are-facebook-digg-and-twitter-so-hard-to-scale.html

[22] J. Rothschild, "High performance at massive scale—Lessons learned at Facebook," 2009 [Online]. Available: http://cns.ucsd.edu/lecturearchive09.shtml#Roth

[23] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998.

[24] H. Kwak, Y. Choi, Y.-H. Eom, H. Jeong, and S. Moon, "Mining communities in networks: A solution for consistency and its evaluation," in *Proc. ACM SIGCOMM IMC*, 2009, pp. 301–314.

[25] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?," 2010.

[26] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *Trans. Knowl. Data Discovery*, vol. 1, no. 1, 2007, Article no. 2.

[27] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," in *Proc. CORR*, 2008, Abs/0810.1355.

[28] M. McGiboney, "Twitter's sweet smell of success," Nielsen, New York, NY, 2009 [Online]. Available: http://blog.nielsen.com/nielsenwire/online_mobile/twitters-tweet-smell-of-success/

[29] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proc. ACM SIGCOMM IMC*, 2007, pp. 29–42.

[30] M. Newman and J. Park, "Why social networks are different from other types of networks," *Phys. Rev. E*, vol. 68, p. 036122, 2003.

[31] M. E. J. Newman, "Modularity and community structure in networks," *Proc. Nat. Acad. Sci.*, vol. 103, p. 8577, 2006.

[32] J. M. Pujol, V. Erramilli, and P. Rodriguez, "Divide and conquer: Partitioning online social networks," 2009 [Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:0905.4918

[33] M. Satyanarayanan, "Coda: A highly available file system for a distributed workstation environment," *IEEE Trans. Comput.*, vol. 39, no. 4, pp. 447–459, Apr. 1990.

[34] F. Schneider, A. Feldmann, B. Krishnamurthy, and W. Willinger, "Understanding online social network usage from a network perspective," in *Proc. ACM SIGCOMM IMC*, 2009, pp. 35–48.

[35] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," in *Proc. ACM SOSP*, 1995, pp. 172–182.

[36] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the evolution of user interaction in Facebook," in *Proc. ACM WOSN*, 2009, pp. 37–42.

Josep M. Pujol received the Ph.D. degree in computer science from the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, in 2006.

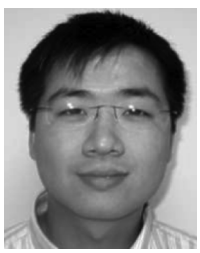
He then went to the University of Michigan, Ann Arbor, for a post-doctoral position and later joined Telefonica Research, Barcelona, Spain, in 2008. He is currently the CTO of 3scale, Barcelona, Spain.

Vijay Erramilli (M'06) received the Ph.D. degree in computer science from Boston University, Boston, MA, in 2008.

He then joined Telefonica Research, Barcelona, Spain, in January 2009, where he is currently an Associate Researcher.

Georgos Siganos received the Ph.D. degree in computer science from the University of California, Riverside in 2005.

He joined Telefonica Research, Barcelona, Spain, in 2007, where he is currently a Researcher.



Xiaoyuan Yang received the Ph.D. degree in computer science from Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, in 2007.

He then joined Telefonica Research, Barcelona, Spain, in 2007.

Nikolaos Laoutaris (M'03) received the Ph.D. degree in computer science from the University of Athens, Athens, Greece, in 2004.

He then went to Boston University, Boston, MA, for a post-doctoral position, and later Harvard University, Cambridge, MA. He joined Telefonica Research, Barcelona, Spain, in 2007 and is currently a Senior Researcher.

Parminder Chhabra received the M.S. degree in computer science from Boston University, Boston, MA, in 2007.

He later joined Telefonica Research, Barcelona, Spain, in 2008.

Pablo Rodriguez (M'04) received the Ph.D. degree in computer science from École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, in 2000.

He is the Director of Research with Telefonica Research and head of the Barcelona Center, Barcelona, Spain. He is also an adjunct faculty member with Columbia University, New York, NY. He has worked with Microsoft Research, Bell Labs, Inktomi, and other startups.