2015 IEEE 35th International Conference on Distributed Computing Systems

Optimizing Inter-Server Communication for Online Social Networks

Jing TANG
Interdisciplinary Graduate School
Nanyang Technological University
Email: tang0311@ntu.edu.sg

Xueyan TANG
School of Computer Engineering
Nanyang Technological University
Email: asxytang@ntu.edu.sg

Junsong YUAN
School of EEE
Nanyang Technological University
Email: jsyuan@ntu.edu.sg

***Abstract*—Distributed storage systems are the key infrastructures for hosting the user data of large-scale Online Social Networks (OSNs). The amount of inter-server communication is an important scalability indicator for these systems. Data partitioning and replication are two inter-related issues affecting the inter-server traffic caused by user-initiated read and write operations. This paper investigates the problem of minimizing the total inter-server traffic among a cluster of OSN servers through joint partitioning and replication optimization. We propose a Traffic-Optimized Partitioning and Replication (TOPR) method based on an analysis of how replica allocation affects the inter-server communication. Lightweight algorithms are developed to adjust partitioning and replication dynamically according to data read and write rates. Evaluations with real Facebook and Twitter social graphs show that TOPR significantly reduces the inter-server communication compared with state-of-the-art methods.**

I. Introduction

Online social networks (OSNs) are popular websites through which huge amounts of people communicate and share information. The most famous OSNs today include Facebook, QQ, Weibo, Google+, Twitter, Renren, LinkedIn, and Flickr. According to Nielsen's latest report [1], people spent 20% of their PC time and 30% of their mobile time on OSNs. No other category of websites is comparable with OSNs in terms of time consumption.

The data maintained by OSNs increase rapidly with their user base. To cope with explosive data growth, a natural solution is to partition the data among a group of servers [2]. Apache Cassandra [3], which combines Amazon Dynamo's consistent hashing scheme [4] and Google BigTable's data model [5], is a distributed data storage system most widely used to support large-scale OSNs such as Facebook and Twitter. However, Cassandra is far from efficient for OSNs due to their highly correlated data access patterns. An OSN user normally accesses not only her own data but also the data of other closely connected users. In Facebook, for instance, a user often views the status, figures, and videos updated by her friends. This property is known as *social locality*. Cassandra performs random data partitioning and replication that are blind to social locality. As a result, it leads to high inter-server traffic caused by user operations in OSNs [6], [7], [8], [9], [10].

Inter-server communication is an important scalability indicator for distributed data storage. Data partitioning and replication are two inter-related issues affecting the amount of inter-server communication. An intuitive approach to preserve social locality in OSN storage is to replicate each user's data on the servers hosting her connected friends. Several methods such as SPAR [6] and S-CLONE [8] have been proposed to maximize the social locality of replication. While creating a replica can reduce the inter-server traffic for reading data, it also introduces new inter-server traffic for synchronization when the data is updated. Since OSN data are constantly created and edited by users, the write-incurred traffic cannot be neglected compared to the read-incurred traffic, particularly when a high degree of replication is needed. Aggressively striving for maximum social locality in replication does not necessarily optimize the total inter-server traffic. A recent SD^3 mechanism [9] proposes to create replicas only when they save more read-incurred traffic than the write-incurred traffic produced. Nevertheless, it assumes fixed data partitions. To the best of our knowledge, little work has studied minimizing the total read-incurred and write-incurred traffic among a cluster of OSN servers through joint partitioning and replication optimization.

In this paper, we propose a Traffic-Optimized Partitioning and Replication (TOPR) method that performs social-aware partitioning and adaptive replication of OSN data in an integrated manner. Based on an analysis of how replica allocation affects the inter-server communication, we develop algorithms to adjust the replicas dynamically according to data read and write rates. Evaluations with the Facebook and Twitter social graphs demonstrate that TOPR can save the inter-server traffic significantly compared with various state-of-the-art methods.

The rest of this paper is organized as follows. Section II reviews the related work. Section III constructs a model of the inter-server traffic and formally defines the optimization problem. Section IV elaborates the design of our TOPR method. Section V presents the experimental setup and results. Finally, Section VI concludes the paper.

II. Related work

Pujol *et al.* [6] proposed a middleware called SPAR to scale OSNs with a cluster of servers. SPAR aims to minimize the number of replicas required for connected users to always have their data co-located on the same servers while maintaining load balance across servers. The perfect social

1063-6927/15 $31.00 © 2015 IEEE
DOI 10.1109/ICDCS.2015.30

locality of SPAR's replication eliminates the need for servers to acquire data from one another upon read requests. Similarly, S-CLONE method developed by Tran *et al.* [8] seeks to maximize the social locality given a fixed number of replicas to set up. However, replicating data at the servers that rarely read it could introduce more write-incurred traffic than the read-incurred traffic saved. Thus, SPAR and S-CLONE are not able to minimize the total inter-server traffic.

Jiao *et al.* [7], [10] studied OSN data placement across multiple geo-distributed clouds (datacenters) for optimizing a range of different objectives. Our work differs from these studies in that we consider data partitioning and replication among a cluster of servers within one cloud (datacenter). In the case of multiple clouds, due to elastic cloud resources, there is practically no limit on the amount of user data that can be hosted by a cloud. However, in the case of a server cluster, each server has a physical capacity limit. To guarantee the service performance, the servers should be prevented from becoming overloaded. Furthermore, the above studies either conduct replication for perfect social locality [7] or create a fixed number of replicas for each user [10]. As discussed above, neither strategy is able to minimize the inter-server traffic. Our proposed TOPR method complements the multi-cloud techniques and they can be combined to maximize the efficiency of OSN services.

SD^3 is a selective data replication mechanism proposed by Liu *et al.* [9] for distributed datacenters. The mechanism avoids replicating the data with low read rates and high write rates to reduce the inter-datacenter traffic. However, SD^3 assumes static data partitioning across datacenters and does not optimize it to further save the traffic. SD^3 also considers different data types separately for replication, such as wall posts and photo comments, due to their different read and write rates. Our proposed method is orthogonal to the data granularity for replication. Separate consideration of different data types can be applied together with our method.

Also relevant to our problem are graph partitioning algorithms [11], [12] and community detection algorithms [13], [14]. The former targets at minimizing the inter-partition edges and the latter aims to find the communities in OSNs. However, these algorithms are offline and cannot cope with the dynamics in OSNs. They are not able to produce stable partitions or communities even upon slight changes to OSN graphs. Chen *et al.* [15] designed community detection algorithms to minimize the inter-server communication of explicit interactions based on the self-similar structure of interaction graphs. However, they did not study any data replication and did not consider the latent interactions which account for the majority of user interactions in OSNs [16].

III. Problem formulation

A. System Model

We consider an OSN service comprising a cluster of servers that store user data. Each user has one *master replica* of her data stored in her *master server* and possibly multiple *slave replicas* stored in some *slave servers*. A user's read and write requests are always sent to her master server. When a user's data is updated, her master server would propagate the update to all of her slave servers for synchronization. When a user u reads the data of another user v, if u's master server does not have v's data, u's master server would fetch the data from a replica of v and then return the result to u. This is known as the relay model in distributed data access [17].

We model the connections between users in an OSN by a social graph $G = (V, E)$, where the set of nodes V represent users and the set of edges E represent the connections among users (e.g., friendships on Facebook, followships on Twitter). Without loss of generality, we assume that the social graph is directed. The existence of an edge (u, v) does not necessarily imply that an edge (v, u) also exists. For each directed edge $(u, v) \in E$, v is called a *neighbor* of u, and u is called an *inverse neighbor* of v. Let $\mathcal{N}_u$ denote the set of user u's neighbors, i.e., $\mathcal{N}_u = \{v : v \in V, (u, v) \in E\}$. For each user u and each server s, we define a binary variable $M_{s,u}$ to describe whether s is the master server of u. $M_{s,u} = 1$ means that u's master replica is stored in server s and $M_{s,u} = 0$ means otherwise. Similarly, we define another binary variable $S_{s,u}$ to describe whether there is a slave replica of user u stored in server s.

Inter-server communication occurs when a user reads her neighbors' data stored in other servers or when her master server pushes a write update to her slave servers. Denote by $r_{u,v}$ and w_u the rates at which user u reads a neighbor v's data and writes her own data,[1] respectively. We assume that the average data size returned by read operations is ψ_r and the average data size of write updates is ψ_w. Let $\mathcal{S}$ be the set of servers. Then, the total inter-server traffic for all the read and write operations is given by

$$\begin{aligned}\Psi =&\psi_r \cdot \sum_{u\in V} \sum_{v\in\mathcal{N}_u} r_{u,v}\Big(1 - \sum_{s\in\mathcal{S}} M_{s,u}(M_{s,v} + S_{s,v})\Big)\\ &+ \psi_w \cdot \sum_{u\in V} \Big(w_u \cdot \sum_{s\in\mathcal{S}} S_{s,u}\Big),\end{aligned} \tag{1}$$

where the first term represents the read-incurred traffic and the second term represents the write-incurred traffic. In the first term, $\sum_{s\in\mathcal{S}} M_{s,u}(M_{s,v} + S_{s,v}) = 1$ if and only if a replica of user v is stored in user u's master server. Thus, u's read operation on v generates inter-server traffic if $\sum_{s\in\mathcal{S}} M_{s,u}(M_{s,v} + S_{s,v}) = 0$. In the second term, $\sum_{s\in\mathcal{S}} S_{s,u}$ represents the total number of u's slave replicas. When writes are conducted on u's data, the updates pushed to all her slave servers give rise to inter-server traffic.

B. Problem Definition

In the above model, the master server of a user needs to handle much more workload than the slave servers. Thus, we use the number of users having a server as their master servers as an indicator of the server load. Assume that each server has a capacity limit of μ. Given a set of servers $\mathcal{S}$, we are interested in finding out the optimal data partitioning and replication that

[1] For simplicity, our model does not include the write updates made by a user on her neighbors' data. From the traffic generated perspective, such an operation can be considered as a combination of two operations: a user reading a neighbor's data and the neighbor updating her own data. Our analysis and algorithms can be easily generalized to handle cross-user writes.

produce the minimum inter-server traffic subject to the server capacity constraints. We formulate this problem by a zero-one quadratic program as follows:

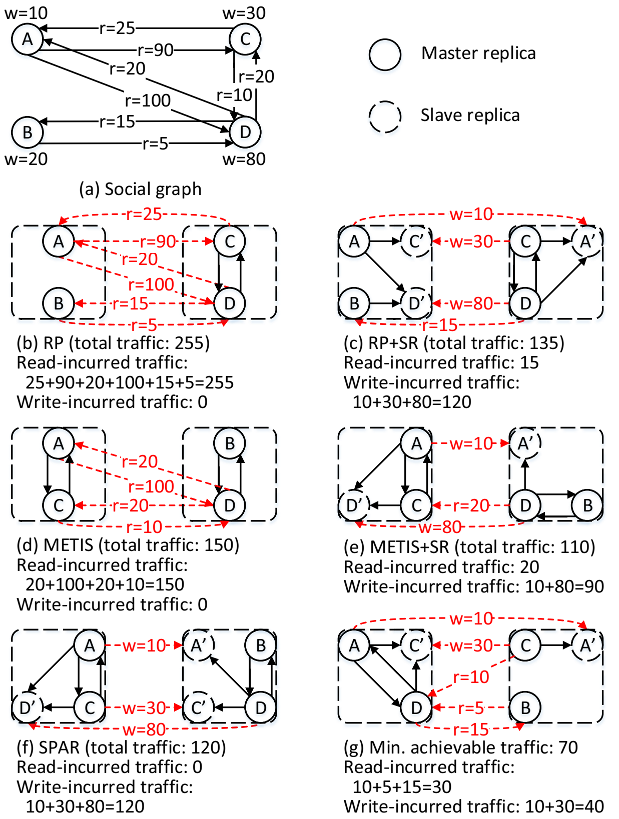
$$
\begin{aligned}
\min \quad & \Psi \\
\text{s.t.} \quad & \sum_{s\in\mathcal{S}} M_{s,u} = 1, \quad \forall u \in V, && (2)\\
& M_{s,u} + S_{s,u} \le 1, \quad \forall u \in V, s \in \mathcal{S}, && (3)\\
& \sum_{u\in V} M_{s,u} \le \mu, \quad \forall s \in \mathcal{S}, && (4)\\
& M_{s,u}, S_{s,u} \in \{0,1\}, \quad \forall u \in V, s \in \mathcal{S}, && (5)
\end{aligned}
$$

where Ψ refers to the total inter-server traffic defined in Eq. (1). Constraint (2) ensures that there exists exactly one master replica of every user. Constraint (3) ensures that each user has at most one replica stored in one server. Constraint (4) restricts each server to host users up to its capacity limit. Constraint (5) regulates the status of the master or slave replica to be either existing or non-existing.

C. Motivation for Joint Optimization

We use a simple example to illustrate the advantage of joint partitioning and replication optimization. Fig. 1(a) shows a social graph with 4 nodes. Each node is marked with a write rate and each edge is marked with a read rate. Suppose that there are two servers available, each having a capacity limit of $\mu = 2$. We compare the inter-server traffic resulting from different methods of partitioning and replication, assuming for simplicity that all the read and write operations involve the data size of one unit.

- RP (Fig. 1(b)) randomly partitions the social graph into two equal size groups and stores them at the two servers without any replication. This is a case of no optimization at all. Since RP does not perform replication, the write operations do not generate any inter-server communication. But a total of 255 traffic units are produced for all the read operations.
- RP+SR (Fig. 1(c)) adds selective replication [9] onto RP by creating replicas only if they can save the inter-server traffic. This is a case of replication optimization without partitioning optimization. RP+SR introduces 120 traffic units for the write operations while reducing the traffic caused by the read operations from 255 down to 15 units. Thus, the total inter-server traffic of RP+SR is 135 units, which is less than RP.
- METIS [11] (Fig. 1(d)) is a graph partitioning algorithm attempting to minimize the total weight of inter-partition edges. Since the edge weights in our social graph represent the read rates, applying METIS to our problem essentially minimizes the inter-server communication assuming no replication is conducted. Thus, this is a case of partitioning optimization without replication optimization. METIS brings down the traffic for the read operations to 150 units and outperforms RP.
- METIS+SR (Fig. 1(e)) adds selective replication onto METIS. This represents a case conducting both partitioning optimization and replication optimization, but the two optimizations are performed *separately*. METIS+SR can further reduce the total inter-server traffic to 110 units.



(a) Social graph

(b) RP (total traffic: 255)
Read-incurred traffic:
25+90+20+100+15+5=255
Write-incurred traffic: 0

(c) RP+SR (total traffic: 135)
Read-incurred traffic: 15
Write-incurred traffic:
10+30+80=120

(d) METIS (total traffic: 150)
Read-incurred traffic:
20+100+20+10=150
Write-incurred traffic: 0

(e) METIS+SR (total traffic: 110)
Read-incurred traffic: 20
Write-incurred traffic: 10+80=90

(f) SPAR (total traffic: 120)
Read-incurred traffic: 0
Write-incurred traffic:
10+30+80=120

(g) Min. achievable traffic: 70
Read-incurred traffic:
10+5+15=30
Write-incurred traffic: 10+30=40

Fig. 1. An example to motivate joint partitioning and replication optimization.

- SPAR [6] (Fig. 1(f)) optimizes partitioning to minimize the number of replicas required for co-locating the neighbors on the same servers. This is again a case of partitioning optimization without replication optimization as slave replicas are blindly created for all pairs of neighbors. Since SPAR guarantees perfect social locality of data storage, no inter-server communication is needed for the read operations, but a total of 120 traffic units are generated by the write operations.
- Unfortunately, none of the above methods achieves the minimum inter-server traffic. The best solution is to optimize partitioning and replication in an integrated manner as shown in Fig. 1(g). Such joint optimization produces the lowest possible inter-server traffic of only 70 units, which is much less than all the earlier methods.

This example shows the importance of optimizing partitioning and replication together. Motivated by this observation, in next section, we develop a Traffic-Optimized Partitioning and Replication (TOPR) method for distributed OSN data storage.

IV. Design of TOPR

The problem defined in Section III-B is NP-hard. Moreover, OSNs are highly dynamic due to constant changes in data access patterns, addition of new users, and creation/deletion of connections. Thus, rather than solving the optimization problem defined in Section III-B statically and offline, in this paper, we focus on online heuristic methods that continuously adapt data partitioning and replication to the dynamics.

A. Preliminaries

We start by studying two basic issues of data partitioning and replication: (1) the optimal allocation of slave replicas given the master replicas; and (2) the change in the inter-server traffic due to the movement of a master replica. They serve as the building blocks in the design of our TOPR method.

Consider a user u. Let s_u denote u's master server. For all the other users whose master replicas are also hosted by s_u, their mutual read accesses with u do not introduce any inter-server traffic. Consider any other server $s \neq s_u$. For each neighbor v of u that is hosted by s, v reads u's data at the rate of $r_{v,u}$. Thus, the aggregate rate of server s reading user u's data is given by

$$R_{s,u} = \sum_{v \in \mathcal{I}_u \cap \mathcal{M}_s} r_{v,u},$$

where $\mathcal{I}_u = \{v : v \in V, (v,u) \in E\}$ is the set of u's inverse neighbors, and $\mathcal{M}_s$ denotes the set of master replicas hosted by s. If there is no slave replica of u on server s, the inter-server traffic between s and s_u due to read operations on u is $\psi_r \cdot R_{s,u}$. If a slave replica of u is created on server s, there would be no read-incurred traffic between s and s_u, but an amount of $\psi_w \cdot w_u$ write-incurred traffic would be introduced between s and s_u for synchronizing the slave with the master. Therefore, to minimize the inter-server traffic, a slave replica of u should be created on server s if and only if

$$\psi_w \cdot w_u < \psi_r \cdot R_{s,u}.$$

It can be seen that the optimal allocation of slave replicas is completely determined by the locations of master replicas, and the optimal allocation can be constructed separately for each user and each server. Algorithm 1 decides whether to create a slave replica of a user u on a server s, where $\mathcal{L}_s$ denotes the set of slave replicas hosted by s.

Algorithm 1: $\boldsymbol{allocateSlave}(u, s)$

```
if $\psi_r R_{s,u} > \psi_w w_u$ then
    $\mathcal{L}_s \leftarrow \mathcal{L}_s \cup \{u\}$;
else
    $\mathcal{L}_s \leftarrow \mathcal{L}_s \setminus \{u\}$;
```

We refer to the traffic between two servers caused by read and write operations on user u as the *u-relevant traffic*. Then, the u-relevant traffic between s_u and s under the optimal allocation of u's slave replicas is

$$\min\{\psi_w \cdot w_u, \psi_r \cdot R_{s,u}\}.$$

Therefore, the total inter-server traffic under the optimal allocation of slave replicas given by

$$\sum_{u \in V} \sum_{s \neq s_u} \min\{\psi_w \cdot w_u, \psi_r \cdot R_{s,u}\}.$$

We now examine the impact of moving a master replica on the inter-server traffic, assuming that the above optimal allocation of slave replicas is implemented both before and after the movement. Suppose that the master replica of a user u is moved from server s_u to another server s. The movement could affect the u-relevant traffic between s_u and s as well as the traffic relevant to u's neighbors involving s_u and s. Specifically, according to the above analysis, prior to the movement, the u-relevant traffic between s_u and s is $\min\{\psi_w \cdot w_u, \psi_r \cdot R_{s,u}\}$. After moving u's master replica to s, the u-relevant traffic becomes $\min\{\psi_w \cdot w_u, \psi_r \cdot R_{s_u,u}\}$. Thus, the reduction in the u-relevant traffic is given by

$$\min\{\psi_w \cdot w_u, \psi_r \cdot R_{s,u}\} - \min\{\psi_w \cdot w_u, \psi_r \cdot R_{s_u,u}\}.$$

For each neighbor v of u, if $s_v \neq s_u$, before moving u's master replica, the v-relevant traffic between s_u and v's master server s_v is

$$\Psi_{s_u,v} = \min\{\psi_w \cdot w_v, \psi_r \cdot R_{s_u,v}\}.$$

After moving u's master replica away from s_u, the v-relevant traffic between s_u and s_v becomes

$$\Psi'_{s_u,v} = \min\{\psi_w \cdot w_v, \psi_r \cdot (R_{s_u,v} - r_{u,v})\}.$$

Similarly, if $s_v \neq s$, before moving u's master replica, the v-relevant traffic between s and s_v is

$$\Psi_{s,v} = \min\{\psi_w \cdot w_v, \psi_r \cdot R_{s,v}\}.$$

After moving u's master replica to s, the v-relevant traffic between s and s_v becomes

$$\Psi'_{s,v} = \min\{\psi_w \cdot w_v, \psi_r \cdot (R_{s,v} + r_{u,v})\}.$$

Thus, the reduction in the v-relevant traffic is given by

$$\begin{cases} \Psi_{s,v} - \Psi'_{s,v} & \text{if } s_v = s_u, \\ \Psi_{s_u,v} - \Psi'_{s_u,v} & \text{if } s_v = s, \\ \Psi_{s,v} - \Psi'_{s,v} + \Psi_{s_u,v} - \Psi'_{s_u,v} & \text{otherwise.} \end{cases}$$

Based on the above analysis, Algorithm 2 calculates the total traffic reduction by moving the master replica of a user u to another server s. The time complexity of Algorithm 2 is given by $O(|\mathcal{N}_u|)$, where $|\mathcal{N}_u|$ is the number of u's neighbors.

Algorithm 2: $\boldsymbol{calMoveMaster}(u, s)$

```
$\delta \leftarrow \min\{\psi_w \cdot w_u, \psi_r \cdot R_{s,u}\} - \min\{\psi_w \cdot w_u, \psi_r \cdot R_{s_u,u}\}$;
for each $v \in \mathcal{N}_u$ do
    if $s_v \neq s$ then
        $\delta \leftarrow \delta + \min\{\psi_w \cdot w_v, \psi_r \cdot R_{s,v}\}$
              $- \min\{\psi_w \cdot w_v, \psi_r \cdot (R_{s,v} + r_{u,v})\}$;
    if $s_v \neq s_u$ then
        $\delta \leftarrow \delta + \min\{\psi_w \cdot w_v, \psi_r \cdot R_{s_u,v}\}$
              $- \min\{\psi_w \cdot w_v, \psi_r \cdot (R_{s_u,v} - r_{u,v})\}$;
return $\delta$;
```

B. TOPR Overview

To optimize the inter-server traffic, we propose a TOPR method that dynamically estimates the read and write rates of the users, and adjusts the allocation of master and slave replicas when these rates change. Algorithm 3 shows the pseudo code of the main algorithm. For each pair of neighbors u and v, we maintain the expected time interval $t_{u,v}$

Algorithm 3: TOPR

```
1  while *True* do
2      if *a read operation is performed* then
3          $(u, v) \leftarrow$ the two users relevant to the read operation ($u$ reads $v$'s data);
4          $\tau \leftarrow$ the time duration since the last read operation of $u$ on $v$;
5          $t_{u,v} \leftarrow \alpha \cdot \tau + (1 - \alpha) \cdot t_{u,v}$;
6          $R_{s_u,v} \leftarrow R_{s_u,v} - r_{u,v} + 1/t_{u,v}$;
7          $r_{u,v} \leftarrow 1/t_{u,v}$;
8          if $r_{u,v}/last\_r_{u,v} \notin [1/\theta_r, \theta_r]$ then
9              $last\_r_{u,v} \leftarrow r_{u,v}$;
10             $\boldsymbol{checkRead}(u, v)$;
11     else if *a write operation is performed* then
12         $u \leftarrow$ the user performing the write operation;
13         $\tau \leftarrow$ the time duration since the last write operation of $u$;
14         $t_u \leftarrow \alpha \cdot \tau + (1 - \alpha) \cdot t_u$;
15         $w_u \leftarrow 1/t_u$;
16         if $w_u/last\_w_u \notin [1/\theta_w, \theta_w]$ then
17             $last\_w_u \leftarrow w_u$;
18             $\boldsymbol{checkWrite}(u)$;
```

between two successive read operations of u on v. Specifically, whenever u performs a read operation on v, we record the time interval τ since her last read operation on v and update the estimate of $t_{u,v}$ using an Exponentially Weighted Moving Average (EWMA) [18] (line 5). The read rate of u on v is then computed as $r_{u,v} = 1/t_{u,v}$ (line 7), and the aggregate read rate of server s_u on v is updated accordingly (line 6). A naive implementation of TOPR is to check for possible replica adjustments that can potentially reduce the inter-server traffic at every read operation. However, this strategy could introduce high computational overheads due to the large volume of user operations. It is intuitive that a slight change in the read rate of a user on a neighbor usually does not deserve any replica adjustment. Thus, to reduce computational overheads, we use a threshold θ_r ($\theta_r \geq 1.0$) to guard the checking for possible replica adjustments. Possible replica adjustments are checked and carried out only if the read rate $r_{u,v}$ has changed relatively by more than a factor of θ_r since the last check (lines 8–10). When θ_r is set to 1.0, the algorithm degenerates to the baseline case that checks for possible adjustments at every read operation. The impact of the guard threshold on the inter-server traffic and computational overheads shall be evaluated in our experiments (Section V-C).

Similarly, for each user u, we maintain the expected time interval t_u between two successive write operations of u by the EWMA (line 14). The write rate of user u is estimated as $w_u = 1/t_u$ (line 15). Possible replica adjustments are checked when w_u has changed relatively by more than a factor of θ_w , where $\theta_w \geq 1.0$ is a guard threshold (lines 16–18).

C. Detailed Design

Algorithm 4 describes how to check and perform replica adjustments upon read operations. When a user u conducts a read operation on another user v, if their master servers are the same, no further action is required since u reading v does not involve any inter-server communication (line 1). Otherwise, we consider the following three possible adjustments: (1) keep the master replicas of u and v unchanged; (2) move the master replica of u to v's master server s_v; and (3) move the master replica of v to u's master server s_u. Note that, in case (1), it may still be possible to adjust v's slave replicas to reduce the inter-server traffic due to the change in the estimate of the read rate $r_{u,v}$. We make use of Algorithm 2 to calculate the traffic reductions of cases (2) and (3) with respect to case (1) (lines 4–7). Recall that the number of master replicas allocated to each server cannot exceed the capacity limit of μ. So, cases (2) and (3) are checked only if the respective servers s_v and s_u have spare capacity to host more master replicas (lines 4 and 6). Finally, the algorithm chooses the adjustment leading to the lowest inter-server traffic and executes the adjustment (lines 8–13).

Algorithm 4: $\boldsymbol{checkRead}(u, v)$

```
1  if $s_u \neq s_v$ then
2      $\delta_2 \leftarrow -\infty$;
3      $\delta_3 \leftarrow -\infty$;
4      if $|\mathcal{M}_{s_v}| + 1 \leq \mu$ then
5          $\delta_2 \leftarrow \boldsymbol{calMoveMaster}(u, s_v)$;
6      if $|\mathcal{M}_{s_u}| + 1 \leq \mu$ then
7          $\delta_3 \leftarrow \boldsymbol{calMoveMaster}(v, s_u)$;
8      if $\delta_2 \geq \delta_3$ *and* $\delta_2 > 0$ then
9          $\boldsymbol{moveMaster}(u, s_v)$;
10     else if $\delta_3 \geq \delta_2$ *and* $\delta_3 > 0$ then
11         $\boldsymbol{moveMaster}(v, s_u)$;
12     else
13         $\boldsymbol{allocateSlave}(v, s_u)$;
```

Algorithm 5: $\boldsymbol{moveMaster}(u, s)$

```
1  $\mathcal{M}_{s_u} \leftarrow \mathcal{M}_{s_u} \setminus \{u\}$;
2  $\mathcal{M}_s \leftarrow \mathcal{M}_s \cup \{u\}$;
3  $\boldsymbol{allocateSlave}(u, s_u)$;
4  $\mathcal{L}_s \leftarrow \mathcal{L}_s \setminus \{u\}$;
5  for *each* $v \in \mathcal{N}_u$ do
6      $R_{s_u,v} \leftarrow R_{s_u,v} - r_{u,v}$;
7      $R_{s,v} \leftarrow R_{s,v} + r_{u,v}$;
8      if $s_v \neq s_u$ then
9          $\boldsymbol{allocateSlave}(v, s_u)$;
10     if $s_v \neq s$ then
11         $\boldsymbol{allocateSlave}(v, s)$;
12 $s_u \leftarrow s$;
```

Algorithm 5 performs the relevant updates when the master replica of a user u is moved to a server s. First, the sets of master replicas hosted by the existing master server s_u and the new master server s are updated (lines 1–2). Then, the optimal allocation of u's slave replica at s_u after the movement is calculated using Algorithm 1 (line 3). Since s has become the new master server of u, u's slave replica at s (if any) is removed (line 4). Due to the change of u's master replica, for

Algorithm 6: $\boldsymbol{checkWrite}(u)$

```
1  $\delta_2 \leftarrow -\infty$;
2  $\delta_3 \leftarrow -\infty$;
3  for each $s \in \mathcal{S} \setminus \{s_u\}$ do
4      if $|\mathcal{M}_s| + 1 \leq \mu$ then
5          $\delta \leftarrow \boldsymbol{calMoveMaster}(u, s)$;
6          if $\delta > \delta_2$ then
7              $\delta_2 \leftarrow \delta$;
8              $s^* \leftarrow s$;
9  if $|\mathcal{M}_{s_u}| + 1 \leq \mu$ then
10     for each $v \in \mathcal{I}_u \setminus \mathcal{M}_{s_u}$ do
11         $\delta \leftarrow \boldsymbol{calMoveMaster}(v, s_u)$;
12         if $\delta > \delta_3$ then
13             $\delta_3 \leftarrow \delta$;
14             $v^* \leftarrow v$;
15 if $\delta_2 \geq \delta_3$ and $\delta_2 > 0$ then
16     $\boldsymbol{moveMaster}(u, s^*)$;
17 else if $\delta_3 \geq \delta_2$ and $\delta_3 > 0$ then
18     $\boldsymbol{moveMaster}(v^*, s_u)$;
19 for each $s \in \mathcal{S} \setminus \{s_u\}$ do
20     $\boldsymbol{allocateSlave}(u, s)$;
```

each neighbor v of u, the aggregate read rates of s_u and s on v should be updated (lines 6–7), and the optimal allocation of v's slave replicas at s_u and s is re-computed as well (lines 8–11).

Algorithm 6 describes how to check and perform replica adjustments upon write operations. When a user u conducts a write operation on her data, we consider the following three possible adjustments: (1) keep the master replicas of u and her neighbors unchanged; (2) move the master replica of u to the master server of one of u's inverse neighbors; and (3) move the master replica of one of u's inverse neighbors to u's master server s_u. Again, in case (1), it may be possible to adjust u's slave replicas to reduce the inter-server traffic due to the change in the estimate of the write rate w_u. For case (2), we use Algorithm 2 to find the best server s^* for hosting u's master replica (lines 3–8). Due to the capacity limit of servers, we consider only the servers that can accommodate more master replicas (line 4). For case (3), we attempt to select the best inverse neighbor v^* of u that would reduce the inter-server traffic most if its master replica is moved to server s_u (lines 9–14). To account for the capacity limit, case (3) is checked only if server s_u has not been filled to its capacity (line 9–14). Finally, the algorithm chooses the adjustment that would result in the lowest inter-server traffic and executes the adjustment (lines 15–20).

D. Other Events

In OSNs, there are several types of events that change the topology of the social graph, including adding and removing nodes (users) and edges (connections). These events are more straightforward to handle than read and write operations. When a new user is created, it does not have any neighbor yet. For the purpose of load balancing, we simply allocate the master replica of the new user to the server hosting the minimum number of master replicas. When an existing user u is deleted, we remove all of u's replicas, including the master and the slaves. Meanwhile, at u's master server, the slave replicas of u's neighbors are adjusted according to Algorithm 1 to account for the removal of the edges incident on u. When a new edge is added from user u to user v, the read rate $r_{u,v}$ is initialized to be 0 since there is no read operation yet. As a result, the new edge would not affect the allocation of slave replicas and no further action is required. When an existing edge from u to v is removed, if their master servers are not the same, v's slave replica at u's master server is adjusted according to Algorithm 1.

E. Discussions

The time complexity of TOPR is mainly determined by that of the routines $\boldsymbol{checkRead}()$ and $\boldsymbol{checkWrite}()$. For $\boldsymbol{checkRead}()$ (Algorithm 4), calculating the potential traffic reductions of cases (2) and (3) by Algorithm 2 takes $O(|\mathcal{N}_u|)$ and $O(|\mathcal{N}_v|)$ time respectively, where $|\mathcal{N}_u|$ and $|\mathcal{N}_v|$ are the numbers of u and v's neighbors. The selected adjustment moves at most one master replica. So, Algorithm 5 takes $O(\max\{|\mathcal{N}_u|, |\mathcal{N}_v|\})$ time to perform the adjustment. Thus, the total time complexity of $\boldsymbol{checkRead}()$ is $O(|\mathcal{N}_u|)+O(|\mathcal{N}_v|)+O(\max\{|\mathcal{N}_u|, |\mathcal{N}_v|\}) = O(|\mathcal{N}_u|+|\mathcal{N}_v|)$. For $\boldsymbol{checkWrite}()$ (Algorithm 6), using Algorithm 2 to determine the best server to host u in case (2) takes $O(|\mathcal{N}_u| \times |\mathcal{S}|)$ time, where $|\mathcal{S}|$ is the number of servers. Selecting the best inverse neighbor of u to move in case (3) by Algorithm 2 takes $O(\sum_{v \in \mathcal{I}_u} |\mathcal{N}_v|)$ time, where $O(\sum_{v \in \mathcal{I}_u} |\mathcal{N}_v|)$ is the number of users sharing an inverse neighbor with u in the social graph. The selected adjustment again moves at most one master replica. Thus, performing the adjustment using Algorithm 5 takes $O(\max\{|\mathcal{N}_u|, \max_{v \in \mathcal{I}_u}\{|\mathcal{N}_v|\}\})$ time. Therefore, the total time complexity of $\boldsymbol{checkWrite}()$ is $O(|\mathcal{N}_u| \times |\mathcal{S}| + \sum_{v \in \mathcal{I}_u} |\mathcal{N}_v|)$. As can be seen, both $\boldsymbol{checkRead}()$ and $\boldsymbol{checkWrite}()$ are lightweight as they involve only the nodes in the immediate neighborhood of u and v.

V. Performance Evaluation

A. Experimental Setup

Two OSN social graphs are selected from [19] to evaluate our algorithms: a Facebook social graph consisting of $4,039$ nodes and $88,234$ edges, and a Twitter social graph comprised of $81,306$ nodes and $1,768,149$ edges. The Facebook graph is undirected and the Twitter graph is directed.

OSN providers seldom publish the data of user activities either for commercial competition or privacy protection purposes. Today, most OSN providers also deploy various mechanisms to defend against crawlers [20]. Thus, it is difficult to get the traces of interactions between OSN users. Two earlier studies [16], [21] used clickstreams to analyze how users interact in OSNs and observed that 92% of user activities on OSNs are profile browsing, which implies that the majority of user interactions are latent interactions. Jiang *et al.* [22] performed a detailed measurement and constructed latent interaction graphs. Similar to other work [10], we use the features reported by these empirical studies to generate user interactions for our simulations.

Specifically, the sets of read rates and write rates for all users are first generated from the power-law distribution with an exponent of 3.5 [22]. Following the statistics reported in [16], the ratio between the total read rate and total write rate is set at $0.92/0.08$. Then, each user is assigned a read rate and a write rate from the above sets. In this process, we control the Spearman's rank correlation coefficient [23] between the read/write rate of each user and her social degree (the number of her neighbors in the social graph), which is set to 0.7 [22]. The write rate assigned to each user represents the rate at which the user updates her data. The read rate assigned to each user represents the aggregate rate at which she reads all of her neighbors. After that, the aggregate read rate is distributed among the neighbors following the preferential model in [24]. That is, the read rate on each neighbor is set proportional to its social degree. After the distribution, the mean read rates on edges are 0.48 per unit time for Facebook and 0.79 for Twitter, and the mean write rates of users are 1.93 for Facebook and 1.66 for Twitter. Finally, we use the Poisson process to generate the sequence of read and write operations for each user according to the assigned rates. We assume an empty social graph at the beginning. The first operation relevant to each user is treated as an event of creating a new user (i.e., adding a node to the social graph). Similarly, the first read operation involving a pair of neighbors is treated as an event of establishing a connection (i.e., adding an edge to the social graph).

The main performance metric used in our evaluation is the total inter-server traffic among a given set of servers. Each read operation is assumed to return a normalized data size of $\psi_r = 1$. The data size ψ_w of each write operation is varied to reflect different ratios between read and write traffic. By default, ψ_w is set to 1. We assume that there are 64 servers available, and all the servers have the same capacity limit. Thus, the minimum requirement of the server capacity is $\lceil\frac{4039}{64}\rceil = 64$ for the Facebook social graph and $\lceil\frac{81306}{64}\rceil = 1271$ for the Twitter social graph. The server capacity limit is set by multiplying the minimum requirement by a factor f ($f \geq 1$) which represents the over-provisioning level of server resources. The larger the factor f, the more flexible the data partitioning among servers. The default value of f is set at 1.0, i.e., the capacity limit is equal to the minimum requirement.

We compare our proposed TOPR method with the methods described in Section III-C.

Random Partitioning: As mentioned earlier, random partitioning is the de facto default distributed storage mechanism for most popular OSNs. We implement the basic method of *random partitioning without replication* (RP), in which no slave replica is created for any user.

METIS: METIS [11] conducts graph partitioning to optimize the inter-server communication assuming no replication. Since METIS is an offline algorithm, it cannot dynamically adapt data partitioning on the fly. In our experiments, we count the numbers of reads and writes in the operation sequence and pre-compute the METIS partitioning using these statistics. The operation sequence is then simulated to measure the inter-server traffic. In this way, our evaluation gives METIS *an unfair advantage* of having a priori knowledge on the data access pattern.

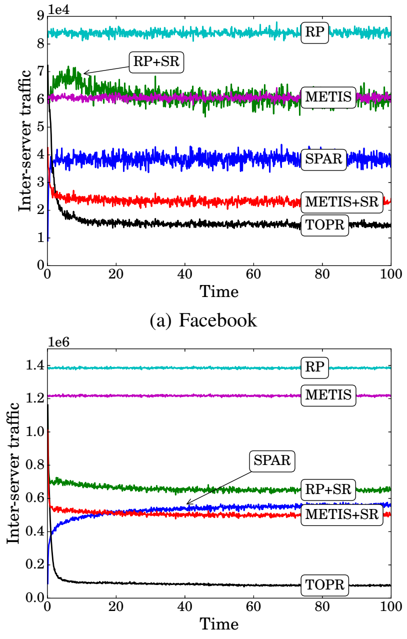
Selective Replication: We apply the selective replication scheme of SD^3 [9] to the data partitions created by the random partitioning and METIS methods. Specifically, slave replicas are created only if they can reduce the total inter-server traffic. The replication decisions are dynamically made using real-time EWMA estimates of read and write rates. The resultant methods are referred to as *random partitioning with selective replication* (RP+SR) and *METIS with selective replication* (METIS+SR).

SPAR: SPAR [6] performs replication with perfect social locality. That is, for each user, a slave replica is stored in the master server of each of her neighbors. SPAR carefully plans data partitioning to minimize the total number of replicas created.

Among the above methods, RP+SR, METIS+SR and TOPR need dynamic estimations of data read and write rates for adjusting replications on the fly. By default, the factor α for EWMA estimations in these methods is set at 0.5. The default guard thresholds θ_r and θ_w for checking possible replica adjustments in our proposed TOPR method are set at 1.0.

B. Comparison of Different Methods

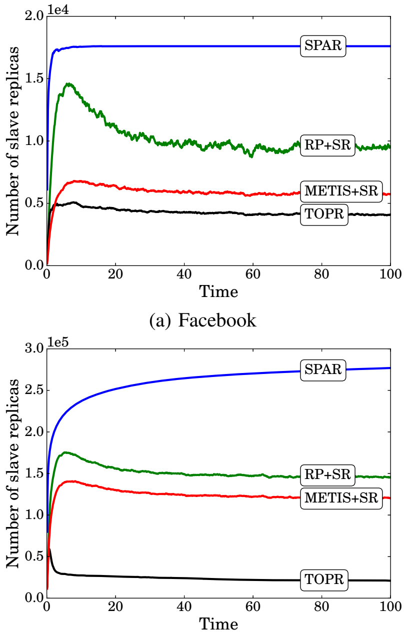
Fig. 2 shows the instantaneous inter-server traffic per unit time produced by different methods under the default parameter settings. The first 10 time units is a warm-up period for most users to join the OSN. After that, among the methods tested, RP produces the highest inter-server traffic as there is no optimization at all. METIS reduces the inter-server traffic significantly compared to RP even though it does not conduct replication either. By creating slave replicas that save more read-incurred traffic than the write-incurred traffic introduced, selective replication considerably cuts the inter-server traffic for different partitioning methods. As shown in Fig. 2, RP+SR and METIS+SR outperform RP and METIS respectively. However, in these two methods, selective replication is carried out separately from partitioning. Our proposed TOPR method optimizes partitioning and replication in an integrated manner, and performs the best among all the methods tested. As seen from Fig. 2, on average, TOPR reduces reduces the inter-server traffic by 75.3% (Facebook) and 87.7% (Twitter) over RP+SR and by 35.5% (Facebook) and 83.9% (Twitter) over METIS+SR. This demonstrates the effectiveness of joint partitioning and replication optimization. The SPAR method, which considers the structure of the social graph in the partitioning and conducts replication with perfect social locality, performs between RP+SR and METIS+SR. This implies that aggressively maximizing the social locality of replication is not very effective for reducing the inter-server traffic.



(a) Facebook

(b) Twitter

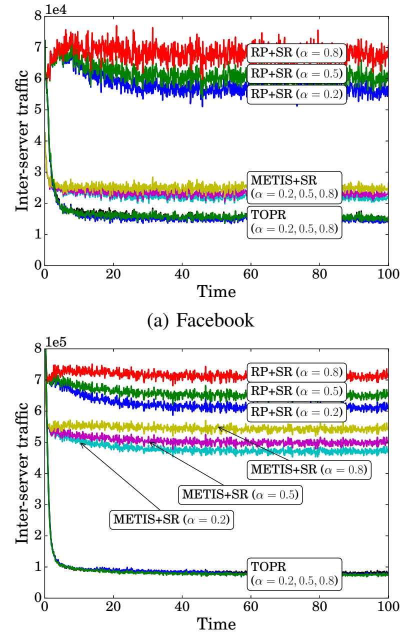
Fig. 2. Inter-server traffic



(a) Facebook

(b) Twitter

Fig. 3. Number of slave replicas



(a) Facebook

(b) Twitter

Fig. 4. Impact of EWMA estimation

TABLE I
AMORTIZED NUMBER OF REPLICA MOVEMENTS PER USER OPERATION.

	RP+SR	METIS+SR	TOPR
Facebook	0.081175	0.027212	0.017224
Twitter	0.054469	0.038767	0.006626

The methods that dynamically adjust replications would produce overheads on the inter-server traffic by moving master/slave replicas based on real-time data access patterns. Table I compares the amortized number of replica movements per read/write operation for RP+SR, METIS+SR and TOPR. As can be seen, the overheads are minor compared to the traffic generated by processing user-initiated read/write operations. In particular, the proposed TOPR method has much lower traffic overheads than RP+SR and METIS+SR. Thus, integrated optimization of partitioning and replication also helps to reduce the traffic overheads.

Fig. 3 compares the total number of slave replicas created by different methods over time. It can be seen that TOPR results in much fewer slave replicas than all the other methods except RP and METIS (which do not perform replication at all). This implies that besides reducing the inter-server traffic, TOPR also significantly decreases the storage cost of replication. Since SPAR maintains perfect social locality of data storage, it creates the highest number of slave replicas.

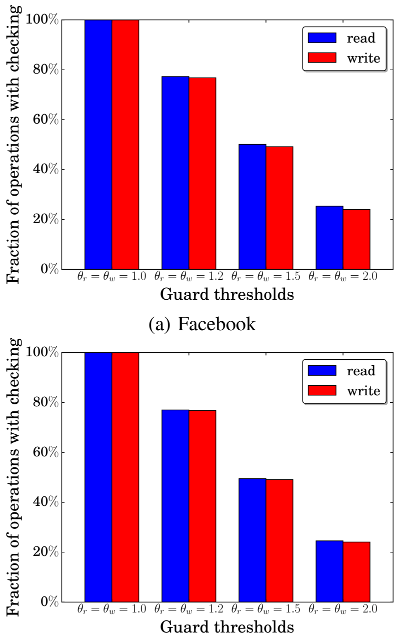
C. Sensitivity of TOPR to Algorithm Parameters

Fig. 4 explores the effect of the EWMA function for estimating the read and write rates. Only the methods with selective replication (RP+SR, METIS+SR and TOPR) make use of the estimated read and write rates to dynamically adjust data replications. We evaluate their performance using three different α values (0.2, 0.5 and 0.8) in the EWMA function. As shown in Fig. 4, the performance variation of each method is within 20% over different α values. This implies that these methods are not very sensitive to α. The relative performance of these methods keeps similar for different α values.

Next, we study the impact of TOPR's guard thresholds θ_r and θ_w. Fig. 5 shows the proportions of read and write operations at which checks are performed when θ_r and θ_w are varied from 1.0 to 2.0. It can be seen that even small guard thresholds can reduce the number of checks significantly. For example, $\theta_r = \theta_w = 1.2$ reduces the number of checks by more than 20% compared to that of $\theta_r = \theta_w = 1.0$. Larger thresholds of $\theta_r = \theta_w = 2.0$ can cut the number of checks by over 75%. Thus, the guard thresholds are useful for reducing the computational overheads of TOPR. Fig. 6 shows the inter-server traffic of TOPR for different thresholds θ_r and θ_w. As can be seen, the threshold values do not affect the inter-server traffic much. Larger thresholds just make it slightly slower for replica allocation in TOPR to converge.

D. Impacts of Various System Parameters

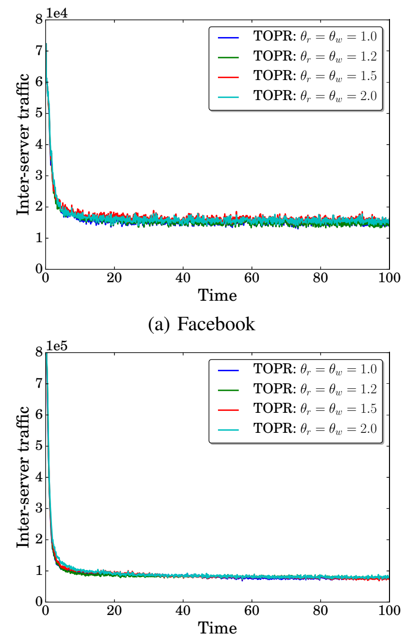
Finally, we study the impacts of various system parameters. In these experiments, we plot the average inter-server traffic per time unit by different methods for comparison. Fig. 7 shows the impact of server capacity limit, where we vary the over-provisioning factor f from 1.0 to 1.5. Larger server capacities allow more users that are strongly connected in the social graph to be allocated the same master server. This reduces the number of slave replicas needed to maintain the social locality of data storage and thus the inter-



(a) Facebook

(b) Twitter

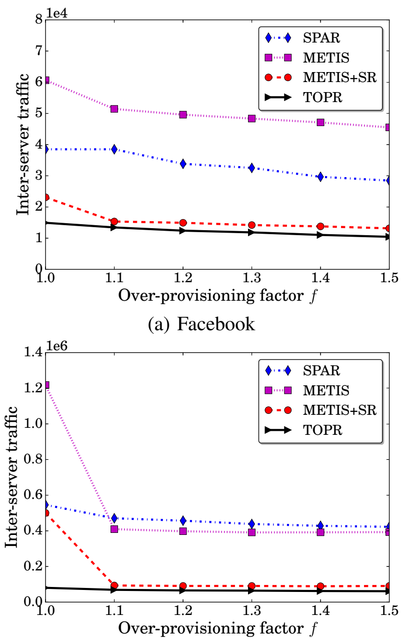
Fig. 5. Workload of checking



(a) Facebook

(b) Twitter

Fig. 6. Impact of guard threshold



(a) Facebook

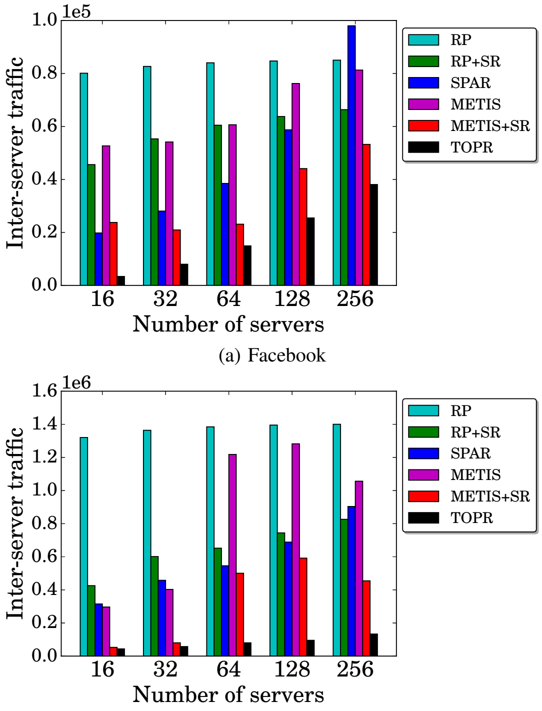
(b) Twitter

Fig. 7. Impact of server capacity

server communication. Therefore, as shown in Fig. 7, the inter-server traffic generally decreases with increasing server capacity for all the four methods that conduct social-aware partitioning (SPAR, METIS, METIS+SR, and TOPR). Our TOPR method consistently outperforms SPAR, METIS and METIS+SR throughout the range of the over-provisioning factor tested. It can also be seen that when the over-provisioning factor is large, the inter-server traffic of METIS+SR can be quite close to that of TOPR. This implies that in such cases, the METIS partitioning together with selective replication may reasonably approximate the optimization problem defined in Section III-B. However, it should be borne in mind that we have assumed a priori knowledge of the data access pattern in computing the METIS partitioning, and METIS is an offline algorithm that is hard to use for real practice.

Fig. 8 shows the inter-server traffic of different methods for different numbers of servers. The inter-server traffic generally increases with the server number. This is because when the number of servers increases, more pairs of neighbors have to be assigned to different servers. As seen from Fig. 8, TOPR always produces less traffic than all the other methods.

Tables II and III report the inter-server traffic of different methods normalized by that of TOPR over a wide range of ψ_w values (the data size of a write operation). Recall that the data size ψ_r of a read operation is fixed at 1. When $\psi_w = 0.01$, the read operations are much more data-intensive than write operations. In this case, selective replication attains nearly perfect social locality in data storage. With nearly perfect social locality, the inter-server traffic of RP+SR, METIS+SR and TOPR is dominated by the write-incurred traffic just like SPAR. Note that SPAR, RP+SR and METIS+SR do not differentiate the users by their write rates in the partitioning. As a result, their inter-server traffic is 1.25 to 9.40 times higher than our proposed TOPR method. On the other hand, RP and METIS, which do not perform any replication, produce



(a) Facebook

(b) Twitter

Fig. 8. Impact of the number of servers

TABLE II
INTER-SERVER TRAFFIC NORMALIZED BY TOPR ON FACEBOOK.

ψ_w	RP	RP+SR	SPAR	METIS	METIS+SR	TOPR
0.01	273.20	8.52	1.25	197.16	1.84	1.00
0.1	33.11	8.51	1.52	23.90	1.89	1.00
1	5.63	4.05	2.58	4.06	1.55	1.00
10	2.33	1.95	10.69	1.68	1.16	1.00
100	1.89	1.94	86.50	1.36	1.22	1.00

TABLE III
INTER-SERVER TRAFFIC NORMALIZED BY TOPR ON TWITTER.

ψ_w	RP	RP+SR	SPAR	METIS	METIS+SR	TOPR
0.01	519.23	9.40	2.05	456.67	6.66	1.00
0.1	79.04	10.00	3.11	69.52	6.98	1.00
1	17.24	8.12	6.79	15.16	6.23	1.00
10	5.91	5.06	23.30	5.20	4.34	1.00
100	3.52	3.56	138.71	3.10	3.13	1.00

the inter-server traffic two orders of magnitude higher than the other methods. When ψ_w increases, less slave replicas are created by selective replication. Hence, the performance gap between RP (METIS) and RP+SR (METIS+SR) demotes. When $\psi_w = 100$, the write operations are much more data-intensive than the read operations. In this case, selective replication loses incentives to create slave replicas. As a result, RP+SR and METIS+SR degenerate to RP and METIS respectively. Their inter-server traffic is dominated by the read-incurred traffic. Since RP+SR does not capture the actual read rates in the partitioning, its inter-server traffic is 1.94 to 3.56 times higher than that of TOPR. METIS+SR performs better than RP+SR, but its traffic is still substantially higher than TOPR. SPAR produces a lot more inter-server traffic than the other methods because it creates a large number of slave replicas to guarantee perfect social locality of data storage. In summary, Tables II and III show that our proposed TOPR method consistently produces much lower inter-server traffic than all the other methods across different intensities of read-incurred and write-incurred traffic.

VI. CONCLUSION

Optimizing the inter-server traffic is a critical issue in the design of distributed data storage systems for OSNs. In this paper, we have formally defined an optimization problem for minimizing the inter-server traffic among a cluster of OSN servers and proposed a method called TOPR to address the problem. TOPR carries out social-aware partitioning and adaptive replication of user data in an integrated manner. Lightweight algorithms are developed for adjusting partitioning and replication on the fly based on real-time data read and write rates. Evaluations with the Facebook and Twitter social graphs show that TOPR not only reduces the inter-server traffic significantly but also saves much storage cost of replication compared to state-of-the-art methods.

ACKNOWLEDGMENT

This research is supported by the Singapore National Research Foundation under its IDM Futures Funding Initiative and administered by the Interactive & Digital Media Programme Office, Media Development Authority. This research is also supported by Academic Research Fund Tier 1 Grant RG29/13.

REFERENCES

[1] Nielsen, "State of the media: The social media report 2012," http://www.nielsen.com/us/en/insights/reports/2012/state-of-the-media-the-social-media-report-2012.html, Online report, December 2012.

[2] D. A. Tran, *Data Storage for Social Networks: A Socially Aware Approach*, ser. SpringerBrief in Optimization Series. Springer, 2012.

[3] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[4] 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," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, Jun. 2008.

[6] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, "The little engine(s) that could: Scaling online social networks," in *Proc. ACM SIGCOMM*, 2010, pp. 375–386.

[7] L. Jiao, J. Li, T. Xu, and X. Fu, "Cost optimization for online social networks on geo-distributed clouds," in *Proc. IEEE ICNP*, 2012.

[8] D. A. Tran, K. Nguyen, and C. Pham, "S-clone: Socially-aware data replication for social networks," *Comput. Netw.*, vol. 56, no. 7, pp. 2001–2013, May 2012.

[9] G. Liu, H. Shen, and H. Chandler, "Selective data replication for online social networks with distributed datacenters," in *Proc. IEEE ICNP*, 2013.

[10] L. Jiao, J. Li, W. Du, and X. Fu, "Multi-objective data placement for multi-cloud socially aware services," in *Proc. IEEE INFOCOM*, 2014, pp. 28–36.

[11] 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, Dec. 1998.

[12] S. Arora, S. Rao, and U. Vazirani, "Expander flows, geometric embeddings and graph partitioning," *J. ACM*, vol. 56, no. 2, pp. 5:1–5:37, Apr. 2009.

[13] M. E. Newman, "Modularity and community structure in networks," *Proc. Nat. Acad. Sci. (PNAS)*, vol. 103, no. 23, pp. 8577–8582, 2006.

[14] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, 2008.

[15] H. Chen, H. Jin, N. Jin, and T. Gu, "Minimizing inter-server communications by exploiting self-similarity in online social networks," in *Proc. IEEE ICNP*, 2012.

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

[17] N. Tran, M. K. Aguilera, and M. Balakrishnan, "Online migration for geo-distributed storage systems," in *Proc. USENIX ATC*, 2011.

[18] S. W. Roberts, "Control chart tests based on geometric moving averages," *Technometrics*, vol. 1, no. 3, pp. 239–250, 1959.

[19] J. McAuley and J. Leskovec, "Learning to discover social circles in ego networks." in *NIPS*, 2012.

[20] M. Mondal, B. Viswanath, A. Clement, P. Druschel, K. P. Gummadi, A. Mislove, and A. Post, "Defending against large-scale crawls in online social networks," in *Proc. ACM CoNEXT*, 2012, pp. 325–336.

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

[22] J. Jiang, C. Wilson, X. Wang, P. Huang, W. Sha, Y. Dai, and B. Y. Zhao, "Understanding latent interactions in online social networks," in *Proc. ACM IMC*, 2010, pp. 369–382.

[23] C. Spearman, "The proof and measurement of association between two things," *The American Journal of Psychology*, vol. 15, no. 1, pp. 72–101, 1904.

[24] I. Hoque and I. Gupta, "Disk layout techniques for online social network data," *IEEE Internet Computing*, vol. 16, no. 3, pp. 24–36, May 2012.