Minimizing Inter-server Communications by Exploiting Self-similarity in Online Social Networks

Hanhua Chen, Hai Jin
Services Computing Technology and System Laboratory
Cluster and Grid Computing Laboratory
School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, 430074, China
{*chen,hjin*}*@hust.edu.cn*

Ning Jin, Tao Gu
Department of Mathematics and Computer Science
University of Southern Denmark
5230 Odense M, Denmark
{*njin,gu*}*@imada.sdu.dk*

***Abstract*—Efficiently operating on relevant data for users in large-scale online social network (OSN) systems is a challenging problem. Storage systems used by popular OSN systems often rely on key-value stores, where randomly partitioning the data of users among servers across the data centers is the defacto standard. Although by using DHTs, the random partition scheme is highly scalable for hosting a large number of users, it leads to costly inter-server communications across data centers due to the complexity of interconnection and interaction between OSN users. In this paper, we explore how to reduce the inter-server communications by retaining the simple and robust nature of OSNs. We propose a data placement solution atop OSN systems to divide users among servers according to the interaction-locality-based structure. Our approach exploits a simple, yet powerful principle of OSN interactions, self-similarity, which reveals that the inter-server communication cost is minimized under such intrinsic structure. Our algorithm avoids a significant amount of inter-server traffic as well as achieves load balance among servers across the data centers.**

We demonstrate the existence of self-similarity in large-scale Facebook traces including 10 million Facebook users and 24 million interaction events. We conduct comprehensive trace-driven simulations to evaluate this design exploiting the unique feature of self-similarity. Results show that our scheme significantly reduces the traffic and latency of the existing schemes.

***Keywords*-Self-similarity, interaction graph, inter-server communications, data center, online social networks**

I. Introduction

Since the emergence of online social networks (OSNs), such as Facebook, Orkut, and Flickr, hundreds of millions of users have started to use OSNs to share information on the Internet. The most popular OSN system, *i.e.,* Facebook, serves more than nine hundred million active users [3] using more than sixty thousand servers located in multiple data centers over the world [1]. Users login, organize events, and search for specific contents. OSNs differ from traditional web applications significantly: they handle highly personalized contents [4], [9] and most importantly they deal with highly interactive operations, producing non-traditional workloads [22].

Currently, popular OSNs commonly utilize scalable key-value stores, where random partition is the defacto strategy for data placement. For example, Facebook utilizes Cassandra [11] as the underlying databases for storing user inbox. Specifically, Cassandra uses a ring infrastructure and consistent hashing [10] to provide high scalability. The consistent hash function assigns each user-id and a server address in the system an m-bit identifier using a base hashing function such as SHA-1. In this manner, a user's data is placed on the first server with the hashing value equaling to or following the hashed identifier of the user. Each server becomes responsible for the region along the ring between itself and its predecessor, so that departure or arrival of a server node only affects its immediate node and other nodes remain unaffected.

Based on the consistent hashing mechanism, friend-scale operations can be very costly. Due to the defacto random partition strategy (such as the default RandomPartitioner option in Cassandra of Facebook) introduced by consistent hashing, friends of a user are assigned to different random servers across the data centers. For example, if a user u has a set of neighbors $\{v_i, i > 0\}$, the data of the neighbors will be placed on different servers according to the hashed key values $key(v_i)$. When user u organizes an event among a number of selected one-hop neighbors, relevant messages will be pushed to all the servers hosting these selected neighbors, raising a large amount of inter-server communications. This can also result in unpredictable response time, determined by the highest latency server. The problem can be particularly acute under heavy data center loads, where network congestion can cause severe network delay. Though the DHT-based scheme is elegant and scalable, random partition of highly interconnected OSN data across the data centers incurs heavy inter-server traffic.

To alleviate the problem caused by the defacto random partition, existing schemes commonly leverage the proper-

978-1-4673-2447-2/12/$31.00

ties of the social graphs. Such schemes assume that social links indicate real user interactions over OSNs. For example, by combining a dynamical partition strategy based on social graph with a user-level replication, Pujol *et al.* [18] achieves local data semantics for all users. A recent research by Wilson *et al.* [22], however, shows that users in OSNs do not interact with most of their friends at all. For the vast majority of Facebook users, 20% of their friends account for 70% of all interactions, implying that social links, and the social graphs they form, are not accurate indicators of information flows between users. Another problem of the partition scheme based on social links in [18] is that the highly dynamic social graph may result in costly replica moving overhead.

Our design philosophy departs from the existing work in such a way that we seek to retain the simple and robust nature of OSNs. Instead of simply leveraging the features of social graphs on OSNs, we have analyzed the interaction graph [22] using a large-scale Facebook trace including 10 million users and 24 million interaction events. We identify a powerful principle: the structure of Facebook interaction graph holds feature of self-similarity, a well known and ubiquitous underlying driving force which minimizes the dissipation of cost/energy in dynamic process [5].

Based on this finding, we propose a novel data placement strategy in OSN systems, which leverages the interaction locality to avoid inter-server interactions while achieving load balance among servers. We conduct comprehensive simulations using real-world traces to evaluate the performance of our design. Results show that our data placement scheme significantly reduces the network traffic and latency of existing schemes.

The main contributions of this work are twofold.

- We identify self-similarity in the structure of Facebook interaction graph, using a large-scale trace including 10 million Facebook users and 24 million interaction events. We also demonstrate that the structure of Facebook social graph has no such feature.
- Based on the observation, we propose a novel data placement strategy for large-scale OSNs, which avoids a significantly large amount of inter-server communications as well as achieves load balance among servers across the data center. Trace-driven simulation results show that the scheme can significantly reduce the inter-server traffic.

The rest of the paper is organized as follows. Section 2 reviews related work. In Section 3, we investigate the principle of self-similarity in Facebook interaction graph. In Section 4, we propose the data placement scheme based on the self-similar structure of interaction graph. In Section 5, we evaluate the performance of this design. Section 6 concludes the paper.

II. Related Work

OSNs are popular infrastructures for user communication and interaction on the Internet, which are also widely used to mitigate email spam [6], improve Internet search [14], and defend against Sybil attacks [21]. Serving hundreds of millions of users, OSN systems are indeed among the Internet's most popular destinations [22].

It is not news that large-scale distributed systems are notoriously difficult to manage. One effective way to optimize the design tradeoffs and guide efficient solution is to understand the system properties which reflects the behavior of users. The recent rapid growth of OSNs has also attracted much interest of research in network structures and properties. Mislove *et al.* [15] present a measurement study on popular OSNs. Their analysis of the social graphs of multiple OSNs demonstrates the power-law, small-world, and scale-free properties of existing OSNs. Their observations reveal that OSNs contain a large, strongly connected core of high-degree nodes, surrounded by many small clusters of low-degree nodes. This suggests that high-degree nodes in the core are critical for the connectivity and the flow of information [9]. Gómez *et al.* [7] conduct statistical analysis on OSNs and show that the degree distributions of OSNs are better explained by log-normal instead of power-law. Leskovec *et al.* [12] introduce the concept of $network\ community\ profile\ plot$ to characterize the quality of a community in OSNs. Newman [16] investigates the property of community structure in large-scale networks and propose an algorithm that uses "edge betweenness" to detect communities. Pujol *et al.* [18] propose to combine a dynamic social graph partition strategy with a user-level replication strategy to achieve local data semantics for users. Most of existing work assumes that social links indicate real user interactions over OSNs.

However, the resent research by Wilson *et al.* [22] shows that the actual interaction activity on Facebook is significantly skewed towards a small portion of each user's social links. The results show that for the vast majority of Facebook users, 20% of their friends account for 70% of all interactions and a user do not interact with most of his friends. Their observation reveals that social links, and the social graphs they form, are not accurate indicators of social relationships between users. Following this observation, in this work, we take a closer look at the user interactions. We identify self-similarity, a principle feature of information flow in large-scale popular OSNs. We further propose a novel data partition scheme to minimize the inter-server communications by leveraging this unique feature.

III. Model

In this section, we present an empirical study of OSN user interactions using real-world traces from Facebook. At first, we briefly introduce the concept of interaction graph [22]. Next, we define a formal mathematical model to

optimize interaction locality by partitioning the interaction graph. Then we measure the traces we collected. At last, we investigate the feature of self-similarity.

A. Interaction Graph

Most of the existing work leverages social graphs to model an online social network. These schemes assume that all social links in the network denote a uniform level of real-world interpersonal association. However, recent analysis on large-scale Facebook interaction traces shows that interaction activity is significantly skewed towards a small portion of each user's social links [22].

To model OSN systems more accurately, Wilson *et al.* suggest to use interaction graph [22]. Different from social graph, where each link represents the relationship between a pair of users, the interaction graph should better differentiate between users' active friends (with recent frequent interactions) and those they merely associate with by name.

Mathematically, an interaction graph of an OSN system is defined as a graph $G = (E, V)$, where V denotes the set of participants and E denotes the set of interaction links with interaction rate greater than a threshold. The threshold is determined by two parameters, n and t, where n denotes a minimum number of interaction events, and t stipulates a window of previous time during which interactions occurred. For example, only the social links reciprocally interact at least once ($n = 1$) in the last year ($t = 1\ year$) are considered as interaction links. In the interaction graph, a user's interaction degree is the number of interaction links associated with him.

The advantages of using interaction graph rather than the social graph are twofold: 1) the interaction graph better represents the user behavior and the information flow across OSNs; 2) the design based on the statistical interaction graph has more stable performance and efficiency than that using the social graph (we will show this in Section 5 using experiment results based on large-scale traces from real-world system). Hence, in this work we model an OSN following the concept of interaction graph [22] and take a closer look at user interactions.

B. Interaction Locality

We use the following symmetric adjacency matrix A with Boolean entries a_{ij} to quantify the interaction links, where e_{ij} is a link with reciprocal interactions performed between participants i and j.

$$a_{ij} = \begin{cases} 1 & \text{if}(e_{ij} \in E) \\ 0 & \text{if}(e_{ij} \notin E) \end{cases} \tag{1}$$

Suppose the interaction graph is partitioned into a set of r interaction communities $\{C_i, 1 \leq i \leq r\}$. We use an $r \times r$ symmetrical matrix whose elements c_{mn} represents the fraction of all edges that link vertices in interaction community C_m to those of C_n,

$$c_{mn} = \frac{\sum\limits_{v_i \in C_m} \sum\limits_{v_j \in C_n} a_{ij}}{\sum\limits_{v_i \in V} \sum\limits_{v_j \in V} a_{ij}} \tag{2}$$

The diagonal elements c_{kk} quantify the fraction of interaction links that fall within C_k,

$$c_{kk} = \frac{\sum\limits_{v_i \in C_k} \sum\limits_{v_j \in C_k} a_{ij}}{\sum\limits_{v_i \in V} \sum\limits_{v_j \in V} a_{ij}} \tag{3}$$

Let δ_i be the fraction of all the ends of the interaction links that are attached to vertices in C_i. We can calculate δ_i straightforward by noting that $\delta_i = \sum\limits_{1 \leq j \leq r} c_{ij}$. If the ends of links are connected together randomly, the expected fraction/probability of the resulting links that connect vertices within C_i is δ_i^2.

We use the following equation to measure the strength of interaction locality [16],

$$Q = \sum_k (c_{kk} - \delta_k^2) \tag{4}$$

Here, $c_{kk} - \delta_k^2$ is the fraction of the edges that fall within C_k minus the expected value of the same quantity if edges fall at random. The larger the deviation is with C_k, the more distinct the interaction locality feature is within C_k. By summing up the deviations of all the interaction communities in the given division, Q quantifies the strength of interaction locality of the entire network. It is not difficult to see that if a particular division does not give more within-community interactions than would be expected by random chance, the model will obtain the minimum value Q = 0. Values other than 0 indicate deviations from randomness.

Hence, by optimizing the division to obtain the maximum value of Q, we can achieve the partition with the strongest interaction locality.

C. Algorithm for Optimizing Interaction Locality

As aforementioned, a high value of Q represents a distinct feature of interaction locality. The division of a given interaction graph can be achieved by optimizing divisions over all possible combinations to obtain the maximal Q. However, exhaustive optimization of Q is very costly. The number of ways to divide n vertices into m non-empty groups is at least exponentially in n. Thus, the optimizing speed is extremely important in designing a practical algorithm, since existing popular OSNs have hundreds of millions of active users. In this paper, we employ a "greedy" algorithm for

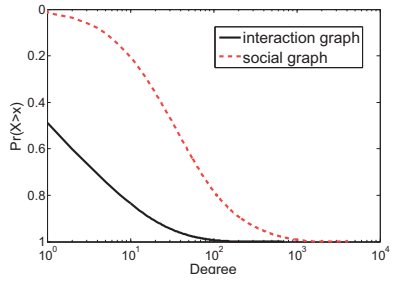

Figure 1. Interaction degree VS. social degree

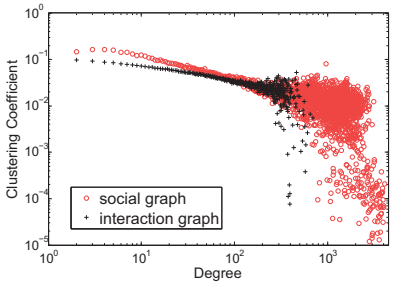

Figure 2. Clustering coefficient

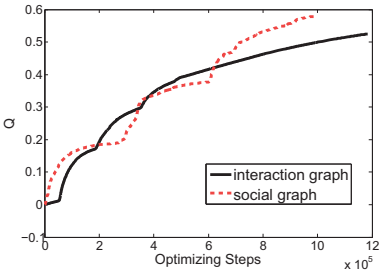

Figure 3. Locality optimization

optimization, which is believed to perform fast over large-scale networks [16].

The pseudo code in Algorithm 1 specifies our optimizing scheme in detail. The algorithm starts with an initial state Γ_0, in which each vertex is a single-member community. At each step, the pair merges if it results in the largest increase in Q. The merge operation continues iteratively until the value of Q converges to the maximum value. Comparing to the other heuristics, the greedy-like approach is more efficient for large-scale networks. In the following, we further speed up the greedy algorithm by taking some optimization strategies.

Looking at each merging step. The change of Q value is $\Delta Q = C_{ij} + C_{ji} - 2\delta_i\delta_j$. In an undirected graph with $C_{ij} = C_{ji}$, we have $\Delta Q = 2(C_{ij} - \delta_i\delta_j)$. It is not difficult to see that joining two interaction communities with no direct edges connecting them results in a negative ΔQ. Therefore, the pairs most likely to join are those with more interaction edges between them, where the interaction locality is preserved most. Consider an interaction graph with m edges and n nodes. Each step costs time $c_1 m$ to find the largest ΔQ, where c_1 is a constant time for computing ΔQ of one pair. When joining two communities, some of C_{ij} should be updated and this time spent will not exceed $c_2 n$, where c_2 is a constant. Thus, each optimizing step takes the worst time of about $c_1 m + c_2 n$. In the worst case, the joining operations will take $n - 1$ steps to achieve the maximum Q, making only one single community left finally. Thus, the worst time cost is $(c_1 m + c_2 n)n$.

In large-scale OSNs, the number of nodes in the complete interaction graph can be hundreds of million. Although current graph analysis platform of OSN company has enough capacity to deal with such large-scale graph by using distributed memory and processing model, the complexities of space and time for algorithm should both be carefully considered for efficiency. According to the real-world traces we collected from Facebook, typically m is ten times larger than n $(i.e., m \gg n)$. Thus, in the worst case, the time complexity is likely to be $O(mn)$, where most of the time of the algorithm is spent on finding the largest ΔQ in each step. By utilizing a heap structure, the time complexity on finding the largest ΔQ in each step can be bounded by $O(\log m)$. Meanwhile, we get extra expenses for joining two communities in order to maintain the heap of ΔQ, the total time cost is now $(c_3 \log m + c_4 n)n$, where c_3 and c_4 are constants different from c_1 or c_2. Though in the worst case, the time spent is still $O(n^2)$ in a sparse graph, the constant coefficient has been greatly decreased since m is many times of n in real-world OSNs.

D. Traces

1) Collection: In this work, we use the interaction traces provided by Ben [22]. The large-scale Facebook trace includes 10 million users and 24 million interaction events from the 22 largest regional networks on Facebook. The large-scale interaction trace is quite representatives for real-world systems.

We have also developed a distributed crawler to collect the social graph traces from Facebook. The distributed crawler written in Java runs in three workstations in parallel each using 100 threads. We have discovered more than two million Facebook users during the period 10 May - 25 June 2011 using three workstations.

2) Analysis: In this section, we analyze the collected Facebook traces. First, we compare the properties of Facebook interaction graph with those of the social graph. We first analyze some general properties including degree distribution, clustering coefficient, and average path length, and then take a closer look at the property of interaction locality. Then, we further investigate the sub-interaction-graph and try to identify the unique feature of the structure of the interaction graph.

Interaction degree. Figure 1 plots the social degree and interaction degree of Facebook traces. The result shows that interaction degree does not scale equally with social degree. Statistically, the average interaction degree of Facebook users is 7.48 while the corresponding number of social graph is 92. It is clear that if all Facebook users interact with each of their friends at least once during the time window, the distribution of the interaction degree is in agreement with the social degree. This, however, is a far cry from the case, indicating that social links mismatch actual relationships.

Clustering coefficient. Clustering coefficient is another measure of degree to which nodes in a graph tend to cluster together. It is defined as the ratio of the number of links that exist between a node's one-hop neighborhood to the exhaustive number of links that could exist. Mathematically,

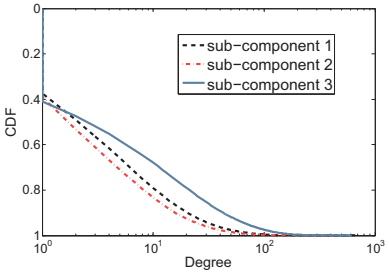

Figure 4. Degrees of sub-graphs

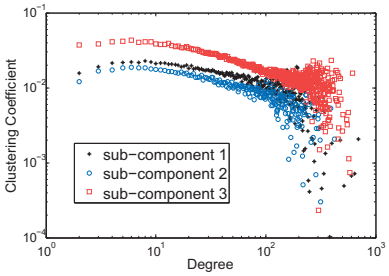

Figure 5. Clustering coefficient of sub-graphs

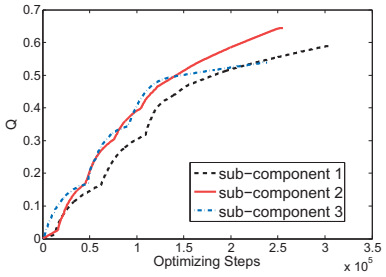

Figure 6. Optimizing procedure of sub-graphs

for a node with m neighbors and e edges between those neighbors, the clustering coefficient can be calculated by: $\frac{2e}{m(m-1)}$. Figures 2 plots the clustering coefficient of the traces. The results show that users with lower degrees have higher clustering coefficients. The results show that the clustering coefficient of the social graph is much higher than that of the interaction graph.

Interaction locality. On top of each trace we collected, we run Algorithm 1 to optimize the interaction locality until the maximum Q is obtained. Figure 3 shows the optimizing process of the fast heuristic algorithm obtaining the best Q using the interaction trace and topology trace. Results show that the best Q values of interaction graph and social graph are 0.578 and 0.524, respectively, which are much larger than those of random graphs. The high values of Q reveal distinct feature of locality in OSN systems. In Fig.3, each curve depicts the whole Q optimizing process from the beginning to the end. Figure 3 also illustrates that each optimizing curve consists of several sub-curves. It is not difficult to see that the gradients of each sub-curve change in a similar way: the slope is very high at the beginning and then drops while the curve becomes flat. The whole curve changes periodically until they reach a max value of Q.

We further examine the optimized division that archives the best Q. The results show that the procedure of Algorithm 1 ends at the state that the interaction graph are divided into hundreds of different interaction communities. When we take a closer look at the communities, we find that the best Q is mainly contributed by several dominating communities. To uncover the properties of the huge components, we conduct further analysis over the three top-components. Figures 4, 5 and 6 plot the degree distribution, cluster coefficient and the best Q for the three largest sub-communities, respectively. The results are in good agreements with the results for the entire network presented in Figures 1, 2 and 3. Here, to investigate the interaction locality feature of these huge sub-communities, we further perform Algorithm 1 over these three huge communities. The procedure returns with best Q values similar with that of the entire interaction graph, indicating these interaction sub-networks also show distinct feature of locality, like their parent.

Such striking similarity between the structures of the entire interaction graph and its sub-graph shown above raises the possibility that the OSN interaction graph might self-organize into a self-similar structure, which is proven to be the underlying driving force to minimize the dissipation of energy/cost during dynamical process [19]. In the following, we make further exploration of the feature of self-similarity using a standard measure.

E. Identifying Self-Similarity in the Interaction Graph

It is not difficult to see that in Algorithm 1 when we obtain the global maximum Q, if we do not stop the merging procedure (this can be easily achieved by repeating merging the pairs which have minimal Q decrease), all the nodes will eventually be merged into a single collection. With this simple extension to Algorithm 1, we consider the entire merging process. It is clear that we can use a binary tree to represent the community merging process. Specifically, at the beginning of the process, there is a collection of individual leaves standing for all the users (single-member communities) in the interaction graph. In the iterative process of Algorithm 1, each merging operation chooses a pair which improves the interaction locality most if the pair merges into a larger community. Thus, the newly formed community can be represented as a bifurcation. With this merging operation continuing, at the end of the process, a binary tree will be formed as shown in Fig.7(a). We call this binary tree the interaction community tree. The root of the tree depicts the whole interaction graph while each of its children represents a sub-structure.

According to our analysis in Section 3.D, the possibility that the interaction graph might self-organize into a form with some cost-oriented quantity optimized is quite appealing and deserves further investigation. To answer this question, we use a standard self-similarity measure for binary-tree structures: the Horton-Strahler (HS) index, originally introduced by Horton and Strahler [20]. Following the HS index measure, consider the binary tree depicted in Fig.7(b). The leaves of the tree are assigned an HS index $i = 1$. For any given branch that ramifies into two sub-branches with HS indices i_1 and i_2, the index is calculated by [20]:

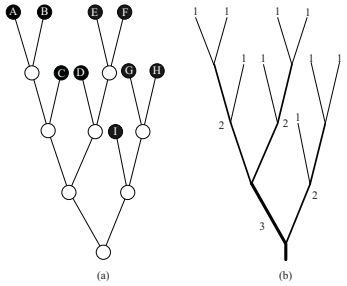

Figure 7. Self-similarity in interaction graph of online social networks

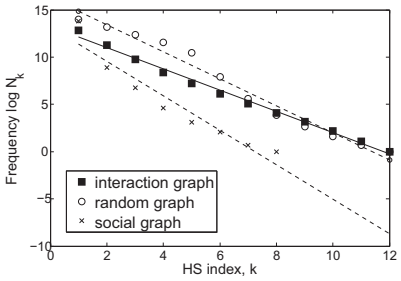

Figure 8. Measurement of self-similarity using HS Index

$$i = \begin{cases} i_1 + 1 & \text{if } i_1 = i_2 \\ max(i_1, i_2) & \text{if } i_1 \neq i_2 \end{cases} \quad (5)$$

Note that the index of a branch changes if and only if it meets a branch with higher or equal index. For interaction communities tree, the interpretation of the HS index is the following. The index of an interaction community changes when it joins an interaction community with same or higher index. Consider, for example, the lowest levels: individuals (with $i = 1$) join to form a group (with $i = 2$), which in turn will join other groups to form a second level group ($i = 3$). Therefore, the index reflects the level of aggregations of interaction communities. Once the HS index of each branch is known, the number of branches N_i with index i can be added up. In the example shown in Fig.7, there are nine branches with index 1 ($N_1 = 9$), three branches with index 2 ($N_2 = 3$), and one branch with index 3 ($N_3 = 1$).

The bifurcation ratios B_i are then defined by [20]:

$$B_i = \frac{N_i}{N_{i+1}} \quad (6)$$

When $B_i = B$ for all i where B is a constant, the structure is determined to be self-similar, because the overall tree can be viewed as being composed of B subtrees, which in turn are composed of B smaller subtrees with similar structures and so forth for all scales [8].

Thus, when self-similarity holds, N_i, the number of branches with HS index i, can be computed according to the function: $N_i = N_1/B^{i-1}$. In Fig. 8, we obtain a fitting of this function for the interaction graph, social graph and a random graph, respectively. The results yield excellent agreement for the interaction graph, while for the social graph and the random graph much worse agreement are obtained. We can further calculate the standard deviation δ_{B_i} of the bifurcation ratios B_i, which approximates zero if self-similarity holds. By comparing δ_{B_i} of Facebook interaction graph with that of the social graph and the random graph with same sizes, we conclude that the interaction graph of the Facebook is self-similar with $B = 3.29$ and $\delta_{B_i} = 0.75$, while both the social graph (with $B = 24$ and $\delta_{B_i} = 46.58$) and random graph (with $B = 4.47$ and $\delta_{B_i} = 3.53$) significantly deviate from self-similarity.

The above standard measure leads us to realize that the community structure of the interaction graph is self-similar. Rinaldo's research shows that such a self-similar structure minimizes the dissipation of energy/cost [19]. As pointed out in a recent paper [5], such scaling properties of networks are ubiquitous. The result in Fig.8 shows that the behavior we observe in the interaction graph in Facebook is in good agreement with the argument [5]. Such an agreement suggests that a common principle of cost optimization could be the underlying driving force in the information flow in popular OSNs. Based on this fingerprint of OSNs, we design a data placement algorithm to minimize the communication cost in OSNs in Section 4.

Algorithm 1 Finding best Q

FindingBestQ(Γ_0)

```
1:  Γ ← Γ_0; Γ_best ← Γ_0, Q ← Q_{Γ_0};
2:  repeat
3:      ΔQ* ← 0;
4:      for ( each pair(C_i, C_j) within Γ ) do
5:          calculate ΔQ_ij ← Q_ij − Q;
6:          /* Q_ij is value after merging C_i and C_j */
7:          if(ΔQ_ij > ΔQ*) then
8:              ΔQ* ← ΔQ_ij;
9:              i* ← i; j* ← j;
10:             /* ΔQ_max ← max{ΔQ_ij}; */
11:         end if
12:     end for
13:     if(ΔQ* > 0) then
14:         Γ ← join_community(C_i*, C_j*);
15:         Q ← Q + ΔQ*;
16:     end if
17: until(ΔQ* < 0)
18: return(Γ);
```

IV. Data Placement Strategy

The objective of our data placement strategy is to minimize the inter-server communication cost by fully utilizing the feature of self-similarity which optimizes the interaction locality. The basic idea is to place the data of the users within an interaction community into the same server. In this way, a significant fraction of operations between users can be handled locally, avoiding a large amount of unnecessary inter-server communications across the data centers. The problem here is that Algorithm 1 may divide the entire interaction

Algorithm 2 Data placement

DivideUsersToServers($CommTree, N, K$)

for $i = 1$ to K **do**
  $Cap[i] \leftarrow N/K$;
end for
$MaxCap \leftarrow max(Cap[i], i = 1 \text{ to } K)$
$MaxServ \leftarrow server$ with $MaxCap$
repeat
  $SubTree \leftarrow$**FindSubTree**($CommTree, MaxCap$)
  Put $users$ in $SubTree$ into $MaxServ$
  Cut $SubTree$ from $CommTree$
  $MaxCap \leftarrow max(Cap[i], i = 1 \text{ to } K)$
  $MaxServ \leftarrow server$ with $MaxCap$
until ($CommTree\text{->}users == 0$)

Algorithm 3 Finding the largest sub-tree

FindSubTree (Tree $root$, **Integer** $capacity$**)**

if ($root\text{->}users \leq capacity$)
  return ($root$)
else
  $left =$ **FindSubTree**($root\text{->}leftChild$)
  $right =$ **FindSubTree**($root\text{->}rightChild$)
  if ($left\text{->}users > right\text{->}users$)
    return ($left$)
  else
    return ($right$)
  end if
end if

Algorithm 4 Placement adjustment

Adjust ($Operation$)

switch $Operation$
  case: AddUser
    $MaxCap \leftarrow max(Cap[i], i = 1 \text{ to } K)$
    $MaxServ \leftarrow$server with $MaxCap$
    Put $Operation\text{->}user$ in $MaxServ$
  case: RemoveUser
    Remove $Operation\text{->}user$
  case: AddLink
    $u1 \leftarrow Operation\text{->}edge\text{->}from$;
    $u2 \leftarrow Operation\text{->}edge\text{->}to$;
    $op \leftarrow mincut$(**move** $u1\text{->}u2$, **move** $u2\text{->}u1$, do nothing);
    execute op
  case: RemoveLink
    for $user$ in ($Operation\text{->}edge\text{->}from$, $Operation\text{->}edge\text{->}to$) **do**
      $MinCut \leftarrow mincut(user\text{->}i, i = 1 \text{ to } K)$
      $MinServ \leftarrow server$ with $MinCut$
      move $user$ to $MinServ$
    end for
  case: AddServer
    $Operation\text{->}Server\text{->}users = 0$
  case: RemoveServer
    $Operation\text{->}Server\text{->}Cap = 0$
    for all $user$ in $Operation\text{->}Server$ **do**
      Adjust(AddUser user);
    end for
end switch

graph into a number of communities with different sizes. In practice, it is not trivial to consider how to achieve load balance when putting data of OSN users to the servers across the data center. For this reason, it is desirable to partition the set of interaction locality preserved communities into a set of servers in the data center, and make each server host similar number of users. We propose the strategy which divides OSN data into the servers across data centers in Algorithm 2 in detail.

The main idea of Algorithm 2 is to cut sub-trees from the interaction community tree (illustrated in Fig. 7) obtained by Algorithm 1, and put all the users on the same sub-tree into the same server. To achieve load balance, we need to assign similar number of leaves to different servers. Here we assume that all the servers have the same capacity. We can easily adapt the scheme to cope with the problem with heterogeneous servers using virtual server techniques. Assume we have a number of K servers in the data center, each having equal capacity, and a number of N users in the network. Algorithm 2 attempts to put a number of $\frac{N}{K}$ users into each server for balancing the load. The division starts from the root. Each time we try to find a sub-tree with a maximum number of leaves which can be placed into a server that has enough capacity to hold the users. This process is repeated until all the leaves are assigned.

The self-similarity principle is indeed the theory foundation of our design. Since the result in Fig.8 shows that the interaction community binary tree is self-similar, each sub-tree cut for user data placement is also self-similar according to the principle of self-similarity. By putting the users in a self-similar sub-tree into a same server in the data center, our data placement strategy would be cost optimized.

In real-world systems, the data placement design should be able to deal with incremental updates/changes of users, interaction links, and number of servers. Instead of re-computing the interaction community tree, we design an incremental adjust algorithm for the dynamic changes in the interaction graph. We describe the detailed adjustment scheme coping with the dynamic changes in Algorithm 3. In Section 5, based on the experiments using the twelve-month real world Facebook user interaction traces, we will further show that the incremental adjustment has very slight influence on system performance and efficiency. Hence, infrequent re-construction of the entire interaction community tree is sufficient for good performance.

V. Evaluation

In this section, we evaluate the performance of our data placement scheme using trace-driven simulations. At first, we describe the details of the simulation setups. Then we compare our scheme with existing schemes. In the evaluation, we use Cassandra [11], Facebook's key-value store as the baseline. We also simulate the little engine scheme recently proposed by Pujol *et al.* [18], and examine the

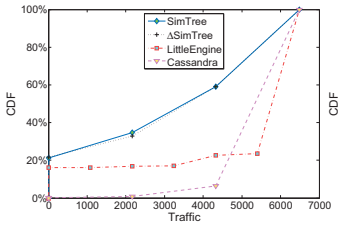

Figure 9. Traffic of wall post

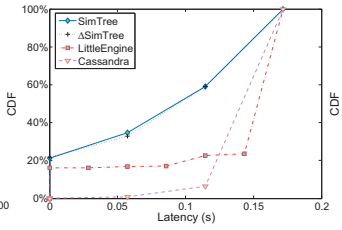

Figure 10. Latency of wall post

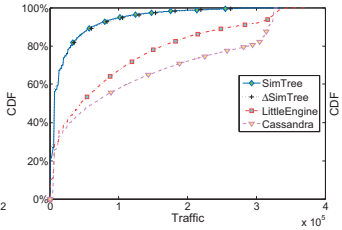

Figure 11. Traffic of profile update

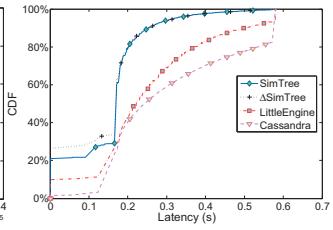

Figure 12. Latency of profile update

performance improvement of our scheme against the little engine scheme.

A. Simulation setups

We simulate both our data placement scheme and previous design including the little engine scheme and the Cassandra strategy using the ns-2 simulator. In order to better represent real-world OSN systems, we consider both the underlying physical data-center network and the logical OSN interaction graph. The physical network should represent the real topology with data center characteristics. Previous studies have shown that the existing large-scale data centers commonly use a fat-tree architecture [17]. A fat-tree built from k-port switches can support non-blocking communications among $\frac{1}{4}k^3$ end hosts using $\frac{5}{4}k^2$ individual k-port switches. The fat-tree as a whole is split into k individual pods, with each pod supporting non-blocking operation among $\frac{1}{4}k^2$ hosts. To simulate the underlying data center networks, we generate a fat-tree network with rich configuration information, including bandwidth configuration, latency, and so forth. For simulating our data placement strategy, we deploy the users into the underlying physical data center servers according to the algorithms we present in Section 4.

In the simulation, we use the traces provided by Ben *et al.* [22], which include ten million Facebook users and 24 million interaction event logs. The large-scale traces collected during more than a whole year well represent the real-world system. We use the interaction traces in the first six months in a year to form the initial interaction graph. Using it, we obtain the interaction community tree to leverage the self-similarly to achieve an initial data placement. We utilize the interaction traces in the second half year for the adjustment of Algorithm 2, where data placement adjustment due to newly added users and interaction links are performed according the the adjust algorithm described in Algorithm 3 in Section 4. In the evaluation, we examine how the dynamic changes in the interaction graph influence the performance and efficiency of our scheme. In the simulation, we first fix the number of data-center servers at 1024. We implement Cassandra and the little engine scheme and compare the performance of our scheme with them. Next, we change the scale of the network to further examine the performance of our design more clearly.

In the simulation, we consider different kinds of operations with three well accepted categories: event-driven, push-on-change, and pull-on-demand [2]. Accordingly, the user operations include wall post, profile update, and user login. We describe the three styles in greater detail as follows.

Event-driven: when a user interacts with selected neighbors, the information should be written to those neighbors. We use this style to simulate the operation of wall post.

Push-on-change: in this style, a status change will be written to all relevant neighbors. We use this style to simulate user profile update. When a user changes his/her key data in the profile, the update should be pushed into his relevant neighbors at once.

Pull-on-demand: when a user needs to read, the system should pull the required data from relevant neighbors. We simulate user login using this style. When a user logins, the system updates his/her home page, which includes all updates from relevant neighbors.

B. Metrics

Our design considers both user-perceived service quality and system efficiency. Quality focuses on the latency, while efficiency focuses on the network traffic in data center. We compute the costs using underlying physical network configurations.

Short latency is always desirable for user operations in OSN systems. Latency for an operation is the sum of the underlying latency over each hop in the data center network. The time required to send a network message includes the propagation time as determined by the link distance between servers and the transmission time determined by the data size and bandwidth capacity in each routing hop. When evaluating the latency we ignore the time spent for local processing, since local data operations are commonly extremely fast due to the fact that most of the operations in Facebook involve only memory access (Memtable), instead of disk access (SSTable) [11].

OSN traffic has a significant impact on the underlying data center network. Heavy traffic limits the scalability of OSN systems. We define the traffic as network resource used for performing an operation, which is mainly a function of the length of the network links, the bandwidths, and other related expenses [13]. Specifically, in the OSN system, when a message is transferred from a user to another, the message may actually traverses a path consisting of a set of underlying physical links in the data centers. The traffic cost is calculated by adding up the cost of the underlying

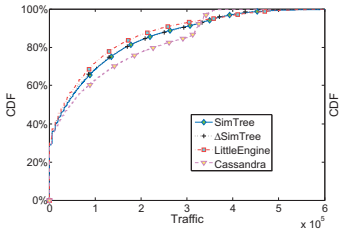

Figure 13. Traffic of users login

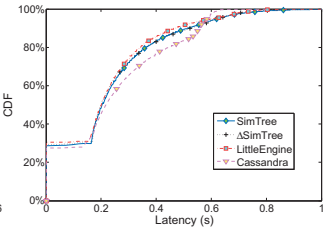

Figure 14. Latency of users login

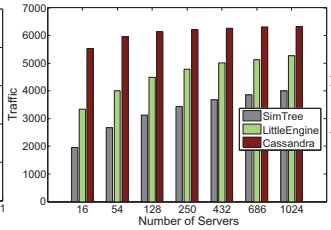

Figure 15. Traffic with # of servers

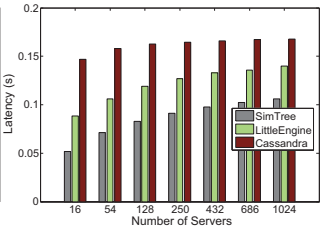

Figure 16. Latency with # of servers

links: $T_c = M \sum_{i=1} \frac{L_i}{B_i}$, where M is the size of the message and L_i and B_i respectively represent the length and the bandwidth of the ith physical link which the message traverses in the underlying data center.

C. Results

Figure 9 plots the traffic of wall post, where 53.2% wall posts using Cassandra have traffic less than 5.4×10^3. By using our data placement scheme leveraging self-similarity, more than 79.48% wall posts have such low traffic, while less than 23.4% of the wall posts using little engine scheme have such traffics. After incremental adjustment, 79.47% of the wall posts using our scheme achieves such a low traffic. The result reveals that our scheme greatly outperform existing schemes, while the dynamic changes of the interaction graph have very slight influence on the performance of our design. In the following, when presenting the results, we use the name SimTree and ΔSimTree to denote our schemes for short, while we use the name little engine to denote the work by Pujol *et al.* [18]. The average traffic of wall posts using Cassandra is 6.4×10^3. The average traffics using our SimTree and ΔSimTree schemes are only 4.0×10^3 and 4.02×10^3, while the average traffic using little engine is 5.3×10^3. The result shows that our scheme significantly reduces the traffic of wall post using Cassandra by 37.5% while reducing that using little engine by 24.5%.

Figure 10 illustrates the latency of wall post, where less than 53.1% of the wall posts using Cassandra have latency less than 0.14 seconds. By using our SimTree and ΔSimTree schemes, more than 79.47% and 79.48% of the wall posts have such short latencies, while about 23.4% of the wall posts using little engine have such latency. The average latency of wall posts using Cassandra is 0.17 seconds. The average latencies of wall posts using our SimTree and ΔSimTree schemes are only 0.1060 and 0.1068 seconds, while little engine scheme needs 0.14 seconds. The result shows that our design significantly reduces the latency of wall post using Cassandra by 37.8%, while reducing that of the little engine scheme by 24.3%.

Figure 11 plots the traffic of profile update operations, where 49.3% profile updates using Cassandra have traffic less than 6×10^4. By using our SimTree and ΔSimTree schemes, more than 89.6% and 89.2% of the profile updates have such low traffics, while only 55.5% little engine profile updates have such low traffic. The average traffic of profile updates using Cassandra is 1.2×10^5. The average traffics using our schemes are only 2.30×10^4 and 2.37×10^4, while the little engine profile updates needs an average traffic of 8.8×10^4. The result shows that our design significantly reduces the traffic of profile updates using Cassandra by 80.3%, while greatly reducing that using the little engine scheme by 73.1%.

Figure 12 shows the latency of profile update, where less than 41.8% of the profile updates using Cassandra have latency less than 0.2 seconds. By using our schemes, more than 81.4% and 81.6% of the profile updates have such short latencies, while only 43.1% profile updates using little engine have such low latency. The average latency of profile updates using Cassandra is 0.32 seconds, while the average latencies using our schemes are only 0.15 and 0.14 seconds. The little engine scheme needs an average latency of 0.26 seconds to update the profile. The results indicates that our design significantly reduces the latencies of profile update operations using Cassandra and little engine by 56.4% and 46.2%, respectively.

Figure 13 depicts the traffic of user login, where 81.5% user login operations using Cassandra have traffic less than 2.5×10^5. By using our data placement scheme leveraging self-similarity, more than 87.9% of the user login operations have such a low traffic, similarly with that of the little engine scheme. After incremental adjustment, 87.7% of the login operations using our scheme has such traffic. The average traffic of Facebook login using Cassandra is 1.0×10^5. The average traffic using SimTree and ΔSimTree are 9.1×10^4 and 8.9×10^4, respectively.

Figure 14 shows the latency of user login operations, where less than 77.1% of the user login operations using Cassandra have latency less than 0.4 seconds. By using our SimTree scheme, more than 83.3% of the user login have such a short latency, while 82.9% of the login operations using ΔSimTree scheme have such a short latency, very similar with that of the little engine scheme. The average latency of user login using Cassandra is 0.25 seconds, while the average latencies using SimTree and ΔSimTree are only 0.23 and 0.22 seconds. The average latency by little engine is 0.22 seconds.

Figure 15 illustrates how the traffic of the overall traces changes with the data center scaling from 16 to 1024 servers. The results show that when the number of servers in the data center increases, the traffic of both the baseline schemes and

our scheme increase. As we use the Facebook trace with fixed scale in the simulation, the increase of the number of servers means that the users are distributed in wider range of data center, causing more inter-server communications. As we can see that the traffic improvements of our scheme and the little engine scheme against Cassandra increases at beginning but decreases in the end. At the point with 16 servers, our scheme significantly reduces the traffic of the little engine scheme by 41.5%, while our scheme reduces the traffic of Cassandra by 64.2%. Figure 16 shows that the latency changes with the data center scaling similarly with that of traffic. At the point with 16 servers, our scheme reduces the latency of little engine scheme by 42.1%, while it reduces the latency of Cassandra by 65.4%.

VI. Conclusion and Future Works

In this work, we demonstrate the existence of self-similarity in the structure of interaction graph of online social networks, revealing the underlying driving force in OSNs which minimizes inter-server communication cost. We further propose a novel data placement scheme based on the self-similar structure of interaction graph which achieves load balance among data center servers while keeping the system operations as local as possible. We evaluate our design using large-scale traces from Facebook and compare our design with existing schemes. Results show that this design can significantly reduce the latency and traffic of operations over large-scale online social networks.

Our model and algorithm for interaction locality can be easily extended to be applied to weighted networks in which each interaction link has a numeric strength associated with it rather than just zero or one. We will investigate how to model the strength of an interaction link in the future work.

Acknowledgment

This work was supported by the grant from the Ph.D. Programs Foundation of Ministry of Education of China (No.20110142120080). We thank Mr. Shaoliang Wu for the help in the simulation.

References

[1] http://www.datacenterknowledge.com/archives/2010/06/28/facebook-server-count-60000-or-more, 2010.

[2] http://highscalability.com/blog/2009/10/13/why-are-facebook-digg-and-twitter-so-hard-to-scale.html, 2010.

[3] http://newsroom.fb.com/content/default.aspx?NewsAreaId=22, 2012.

[4] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin, "Persona: an online social network with user-defined privacy," in *SIGCOMM*, 2009.

[5] M. Buchanan, "A game of chance," *Nature*, vol. 419, p. 787, 2002.

[6] S. Garriss, M. Kaminsky, M. J. Freedman, B. Karp, D. Mazi'eres, and H. Yu, "Re: Reliable email," in *NSDI*, 2006.

[7] V. Gómez, A. Kaltenbrunner, and V. López, "Statistical analysis of the social network and discussion threads in slashdot," in *WWW*, 2008.

[8] T. C. Halsey, "The branching structure of diffusion-limited aggregates," *Europhysics Letters*, vol. 39, no. 1, p. 43, 1997.

[9] M. U. Ilyas, M. Z. Shafiq, A. X. Liu, and H. Radha, "A distributed and privacy preserving algorithm for identifying information hubs in social networks," in *INFOCOM*, 2011.

[10] D. Karger, E. Lehman, T. Leightom, M. Levine, D. Lewin, and R. Panigrahy, "Consisten hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *STOC*, 1998.

[11] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[12] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Statistical properties of community structure in large social and information networks," in *WWW*, 2008.

[13] Y. Liu, J. Han, and J. Wang, "Rumor riding: Anonymizing unstructured peer-to-peer systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 3, pp. 464–475, 2011.

[14] A. Mislove, K. P. Gummadi, and P. Druschel, "Exploiting social networks for internet search," in *HotNets*, 2006.

[15] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *IMC*, 2007.

[16] M. Newman, "Fast algorithm for detecting community structure in networks," *Physical Review E*, vol. 69, p. 066133, 2004.

[17] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: a scalable fault-tolerant layer 2 data center network fabric," in *SIGCOMM*, 2009.

[18] J. 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 *SIGCOMM*, 2010.

[19] A. Rinaldo, I. Rodriguez-Iturbe, R. Rigon, E. Ijjasz-Vasquez, and R. L. Bras, "Self-organized fractal river networks," *Physical Review Letters*, vol. 70, no. 6, pp. 822–825, 1993.

[20] A. N. Strahler, "Hypsometric (area-altitude) analysis of erosional topography," *Bulletin of the Geological Society of America*, vol. 63, pp. 1117–1142, 1952.

[21] B. Viswanath, A. Post, P. K. Gummadi, and A. Mislove, "An analysis of social network-based sybil defenses," in *SIGCOMM*, 2010.

[22] C. Wilson, B. Boe, A. Sala, K. Puttaswamy, and B. Zhao, "User interactions in social networks and their implications," in *EuroSys*, 2009.