Cuckoo: Scaling Microblogging Services with Divergent Traffic Demands.

Univ. of Goettingen Technical Report No. IFI-TB-2011-01

Tianyin Xu*, Yang Chen*, Lei Jiao*, Ben Y. Zhao†, Pan Hui‡, Xiaoming Fu*
*Institute of Computer Science, University of Goettingen, Germany
†Department of Computer Science, University of California, Santa Barbara, USA
‡Deutsche Telekom Laboratories and TU-Berlin, Germany

Abstract

Today's microblogging services such as Twitter have long outgrown their initial designs as SMS-based social networks. Instead, a massive and steadily-growing user population of more than 100 million is using Twitter for everything from capturing the mood of the country to detecting earthquakes and Internet service failures. It is unsurprising that the traditional centralized client-server architecture has not scaled with user demands, leading to server overload and significant impairment of availability.

In this paper, we argue that the divergence in usage models of microblogging services can be best addressed using complementary mechanisms, one that provides reliable messages between friends, and another that delivers events from popular celebrities and media outlets to their thousands or even millions of followers. We present *Cuckoo*, a new microblogging system that offloads processing and bandwidth costs away from a small centralized server base while ensuring reliable message delivery. We use a 20-day Twitter availability measurement to guide our design, and trace-driven emulation of 30,000 Twitter users to evaluate our Cuckoo prototype. Compared to the centralized approach, Cuckoo achieves 30-50% server bandwidth savings and 50-60% CPU load reduction, while guaranteeing reliable message delivery.

I. Introduction

In recent years, microblogging services such as Twitter have reached phenomenal levels of success and become a significant new form of Internet communication utility. Twitter, one of the most successful

service of such, has more than 100 million users and generates more than 65 million "tweets" per day [44], [54]. In addition, Twitter usage peaks during prominent events. For example, a record was set during the FIFA World Cup 2010 when fans wrote 2,940 tweets per second in a 30-second period after a goal [41].

While originally designed as an online social network (OSN) for users with quick updates, usage of Twitter has evolved to encompass a wide variety of applications. Twitter usage is so wide spread and pervasive that its traffic is often used as a way to capture the sentiment of the country. Studies have used Twitter traffic to accurately predict gross revenue for movie openings [22], even producing effective predictions for election results [26]. Still other projects have demonstrated that Twitter traffic can be mined as a sensor for Internet service failures [40] and even a real-time warning system for earthquakes [2].

In addition to these applications, Twitter usage has also evolved significantly over time. Recent studies have shown that while many users still use it as a social communication tool, much of Twitter traffic today is communication from celebrities and well-known personalities to their fans and followers [12], [34]. These asymmetric communication channels more closely resemble news media outlets than social communication channels. Because of the popularity of celebrities on Twitter (e.g., Britney Spears, Lady Gaga, and Justin Bieber account for over 26 million followers), these accounts are generating a large amount of traffic and placing tremendous load on Twitter's servers.

These major sources of traffic have a very tangible impact on the performance and availability of Twitter as a service. Despite their efforts to scale the system, Twitter as a service has suffered significant loss in availability from malicious attacks and hardware failures [14], [21], and more frequently from traffic overload and flash crowds [48], [55], [56]. As short-term solutions, Twitter has employed per-user request and connection rate limits [25], as well as network usage monitoring and doubling the capacity of internal networks [56], all with limited success. Given the rapid and continuing growth of traffic demands, it is clearly challenging and likely costly to scale up with the demands using the current centralized architecture.

In this paper, we explore an alternative architecture for popular microblogging services such as Twitter. In our system, *Cuckoo*[1], our goal is to explore designs that leverage bandwidth and processing resources at client machines without sacrificing service availability or reliable delivery of contents. One of our insights is to recognize the two different roles these services play, those of an online social network and a news delivery medium. We use complementary mechanisms to address the dual roles while minimizing resource consumption. In the social network component, users are connected via mostly symmetric social links,

[1]We first outlined our idea in an earlier workshop paper [60].

and have a limited number of connections. Here, we allow a "publisher" or creator of a tweet to directly push the content to his (or her) friends via unicast. In the news delivery component, content producers are typically celebrities or media outlets, each connected via asymmetric links to a large number of followers. Given the large number of users with shared interests, we use gossip to provide highly reliable and load-balanced content delivery. Moreover, Cuckoo's delivery mechanisms support heterogeneous client access (e.g., mobile phone access) which is becoming increasingly common in microblogging services [39].

To ensure consistency and high data availability, Cuckoo uses a set of centralized servers to augment client peers in a peer-assisted architecture. This combination greatly simplifies data management challenges while reducing the server load. From an economic perspective, a Cuckoo service provider is still viable, because he (or she) will keep the master copies of all user contents, and can still generate revenue e.g., by using content-based ads.

We have implemented a Cuckoo prototype and made its source code and datasets publicly available[2]. We evaluated a small-scale deployment for 50 Twitter users running on 5 laptops as a demonstration [59]. In addition, we have conducted laboratory experiments using a detailed Twitter trace containing over 30,000 users. We show that Cuckoo incurs 30-50% server bandwidth savings, 50-60% server CPU reduction compared with its centralized ilk, as well as reliable message delivery and efficient micronews dissemination (over 92.5% coverage) between Cuckoo peers.

In summary, this paper makes three key contributions:

1) A novel system architecture tailored for microblogging services to address the scalability issues, which relieves main server burden and achieves scalable content delivery by decoupling microblogging's dual functionality components.
2) A detailed availability measurement of Twitter during a flash crowd event.
3) A prototype implementation and trace-driven emulation of 30,000 Twitter users yielding notable bandwidth saving, CPU and memory reduction, as well as reliable message delivery and efficient micronews dissemination.

The remainder of the paper is organized as follows. Section II overviews the background and related work. Section III demonstrates the motivation based on our measurements. We propose Cuckoo's system architecture and design rationales in Section IV. In Section V, we analyze the results of experiments conducted to evaluate Cuckoo using the Twitter trace. Section VII concludes the paper with final remarks.

[2]Cuckoo source code and selected datasets can be found at http://mycuckoo.org/.

II. Background and Related Work

With immense and steadily-growing popularity over recent years, microblogging services have attracted considerable interests in the research community. We provide some background and summarize the state of the art.

A. Microblogging Model

Although different services offer different additional features, common to microblogging is the opt-in subscription model based on the "follow" operation. The microblogging model is deceptively simple: A user can publish tweets within a length limit of viewable text (e.g., up to 140 characters in Twitter). The other users who have explicitly followed that user will receive his (or her) tweets, i.e., being a *follower* means that the user will receive all the tweets from his (or her) *followees*. Currently, the microblogging model is implemented by using naïve polling for detecting updates in a centralized architecture.

There are some prior works on RSS feed systems that abandon the use of naïve polling and achieve high scalability and performance [42], [49]. Their key idea is cooperative polling between dedicated brokers. Microblogging differentiates from RSS by the system architecture. In microblogging, there is no always-on broker that collects events from publishers and sends feeds to subscribers. The key problem of microblogging is how to directly deliver publishers' tweets to their followers. Cuckoo shares the insight with these works that the blind polling is the prime culprit of poor performance and limited scalability. Instead, Cuckoo enables users to sharing tweets in a peer-assisted fashion. On the other hand, Cuckoo interoperates with the current polling-based web architecture, requiring no change to legacy web servers.

B. Publish-Subscribe System

The publish-subscribe (pub-sub) interaction paradigm [19] mainly consists of three components [42]: (1) *Publishers* who produce and feed the information into the system; (2) *Subscribers* who consume the information that matches their registered interests; (3) *Middleware mediators* who match the interests of subscribers with the information produced by publishers, and then deliver the notification or information to subscribers. To provide scalability, the middleware mediators are usually organized as overlay networks. Based on the ways of specifying the events of interests, pub-sub systems can be classified into three categories [19]: *topic-based* systems (e.g., TIBCO [1], Isis [3], Herald [8]), *content-based* systems (e.g., Siena [9], Astrolabe [57], XTrie [13], YFilter [17]), and *type-based* systems [18]. Content-based and type-based pub-sub systems have much higher degree of interest expressiveness than the topic-based systems

that are static and primitive. Meanwhile, the content-based and type-based systems require sophisticated in-network filtering and aggregation structures that cause much higher runtime overhead.

The microblogging model can be seen as a simplified topic-based publish-subscribe system. We call it the *identity-based* pub-sub system because users explicitly subscribe (i.e., follow) other user identities and do not need content filtering/matching, but receive all the updates from their followees. Moreover, in the microblogging systems, there is no middleware mediator (i.e., broker) to help aggregate contents and disseminate notifications, i.e., the interactions between followers and followees are direct and *ad hoc*. Different from previous research efforts on pub-sub systems that primarily focus on enhancing content filtering/matching, Cuckoo aims at dealing with the divergent traffic demands of microblogging services which have not been observed in traditional pub-sub systems.

C. Microblogging Measurement and Analysis

Microblogging services are widely recognized as online social network services for the explicit and implicit social relations [25], [29], [32], [40]. For example, users exhibiting reciprocity (i.e., following each other) should be acquaintances, typical in OSNs [37]. According to the "follow" relations, Krishnamurthy *et al.* identify distinct groups of users, e.g., broadcasters and evangelists [32]. Different social groups have different social behavior. Ghosh *et al.* study the relations and restrictions on the number of social links in microblogging, based on which a network growth model is proposed [25]. Java *et al.* report early observations of Twitter and analyze social communities formed by users with similar interests [29]. On the other hand, some researchers recently argue that microblogging, as exemplified by Twitter, serves more as news media outlets than OSN services [12], [34], [50]. Due to the one-sided nature of the "follow" relation, there are a small number of highly-subscribed users (e.g., celebrities, mass media) who have large numbers of followers and post far more tweets than other users. These users generate the greatest per-capita proportion of network traffic and trend the trends. In fact, microblogging has already been used as an important media for news breaking and product marketing [2], [28].

One of Cuckoo's design rationales is to separate the dual functionality components of microblogging, i.e., social network and news media. Cuckoo employs different mechanisms towards scalable message delivery, gearing to the different dissemination models of the two components. Moreover, Cuckoo takes advantage of the inherent social relations to optimize system performance and information sharing.

D. Decentralized Microblogging and OSN Systems

There are a number of decentralized OSN prototypes proposed for different research concerns. FETHR [50] is a recently proposed microblogging system that envisions fully decentralized microblogging services. Its main idea is to let users directly contact each other via HTTP and employ gossip for popular content propagation. However, as a truly P2P system, FETHR cannot guarantee reliable data delivery since it does not consider the asynchronism of user access. As a result, some tweets will not get to users. Moreover, FETHR does not elaborate the gossip dissemination component nor implement it in its prototype. Other practical issues such as client heterogeneity support and user lookup are also missing in FETHR.

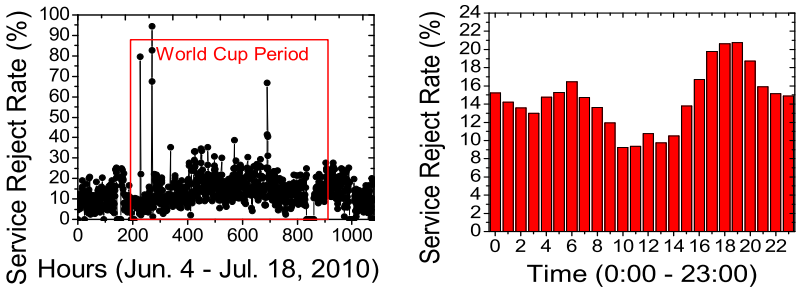
PeerSoN [7] is a project for P2P OSNs that uses encryption to protect user privacy against OSN providers and third-party applications. PeerSoN uses dedicated DHT for meta-data lookup. Based on the meta-data, direct information exchanging between users can be achieved. Vis-à-Vis [52] is based on the concept of Virtual Individual Servers (VISs), which is a kind of paid cloud-computing utility such as Amazon EC2 used for managing and storing user data. VISs self-organize into multi-tier DHTs that represent OSN groups, with one DHT per group. The top-tier DHT is used to advertise and search for other groups. Safebook [15] is a decentralized OSN that aims at protecting users' security and privacy. In Safebook, client nodes are organized into "matryoshka" structures to protect user data based on trust transitivity. uaOSN [33] proposes to distribute only the storage of heavy contents while retaining the business model of OSN providers.

Cuckoo proposes a new system architecture tailored for microblogging services. It consists of two overlay networks based on which different content delivery mechanisms are employed. The delivery mechanisms support heterogeneous client access by differentiating client types. On the other hand, since fully distributed P2P systems have hardly achieved success in terms of availability and reliability, Cuckoo employs a set of servers as a backup database that ensures high data availability and effectively eliminates the inconsistency due to the asynchronism of user access.

III. Measuring Availability at High Load

To provide concrete motivation for our work beyond the prior efforts, we conducted measurement studies on current microblogging systems. Our study includes a 20-day Twitter availability and performability[3] measurement and a user behavior analysis for over 300,000 Twitter users. In addition, we measure the system scalability of a generic centralized microblogging architecture.

[3]We simply call "performability" a measure of performance after faults or flash crowds, i.e., the "ability to perform" [6].



(a) Average service reject rate per hour in the measurement period

(b) Average service reject rate for each hour in one day

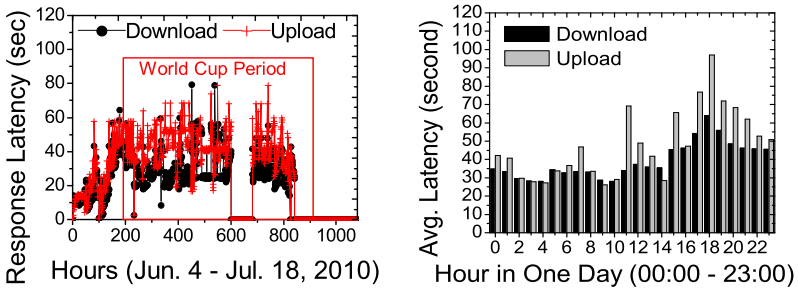
Fig. 1. Service reject rate

A. *Availability and Performability of Twitter*

We first conducted a measurement study on the availability and performability of Twitter in terms of service reject rate and response latency. The study was set in NET lab in Göttingen, Germany from 00:00, Jun. 4 to 23:00, Jul. 18, 2010, Berlin time (CEST), including the period of World Cup 2010 in the same time zone, which is regards as Twitter's worst month since October 2009 from a site stability and service outage perspective [55]. We used JTwitter as the Twitter API to do the measurement.

For the service reject rate, we randomly selected a Twitter user and sent the request for his (or her) recent 200 tweets to the Twitter site every 5 seconds. If the Twitter site returns a 50X error (e.g., 502 error), it indicates that something went wrong (over-capacity in most cases) at Twitter's end and we count for one service reject event. Fig. 1(a) shows the average service reject rate per hour during our measurement period. We can see that Twitter's availability was poor – the reject rate was already about 10% in normal time. Moreover, the flash crowd caused by FIFA World Cup had an obvious impact on service reject rate which increased from 10% to 20%. Since the flash crowd generated a significant surge over Twitter servers' capacity, the performance of the offered service degraded tremendously. Fig. 1(a) reports the average reject rate for each hour in one day. We find that there existed some peak hours (e.g., 18:00 – 19:00) that had the worst performance in terms of service reject.

For response latency, we measured both upload latency and download latency. Upload latency refers to the interval between sending a tweet to the Twitter site and receiving the ACK, while download latency is the interval between sending the request and receiving the required contents. For one measurement round, we first generated an artificial tweet by combining random characters in an predefined alphabet, posted it on Twitter and recorded the upload latency. Then, we requested the posted tweet from Twitter and recorded the download latency. Such round repeated every 5 seconds. Similar as Fig. 1, Fig. 2 shows



(a) Average response latency per hour in the measurement period

(b) Average response latency for each hour in one day

Fig. 2. Response latency for {up, down}load requests

the measured response latency of the Twitter service[4]. No surprisingly, Twitter's performability, in terms of response latency, was unsatisfying especially during World Cup with the download latency about 200 seconds and upload latency about 400 seconds. Twitter engineers also noticed this outage and poor site performance [55], [56], their solutions include doubling the capacity, monitoring, and rebalancing the traffic on their internal network, which do not scale well with the unprecedented growth.

B. User Access Pattern Analysis

To further study the server load according to user access patterns of Twitter services, we analyze large-scale Twitter user traces. We collected 3,117,750 users' profile, social relations, and all the tweets maintained on the Twitter site. Using 4 machines with whitelisted IPs, we used snowball crawling that began with the most popular 20 users reported in [34] using Twitter API. The crawling period was from Mar. 6 to Apr. 2, 2010. In this paper, we focus on the user access patterns in the one-week period from Feb. 1 to Feb. 7, 2010. To simplify the problem and yet accurately represent the traffic patterns of Twitter services, we consider two built-in Twitter's interaction models: *post* and *request*. For session durations, we use the duration dataset provided in [27]. The polling period is set as one minute according to the setting options of common Twitter clients (e.g., Ambientweet, Gwibber, Osfoora)[5]. The details of the datasets and data processing are described in Section V-A.

Fig. 3 shows the time-series server load in terms of the number of received messages on the server side. We can see that over 90% are request messages which make up the dominating traffic proportion.

[4]The gaps in Fig. 2(a) is due to server cutoffs during the measurement period.

[5]Most existing Twitter client software allows users to set the polling/updating interval from 1 second to 600 seconds (e.g., Osfoora), or provides options like 1 min, 2 min, ..., 5 min (e.g., Ambientweet). Since there is no standard polling interval, we use one minute as a representative considering the rate limit of Twitter which allows 150 requests per hour.

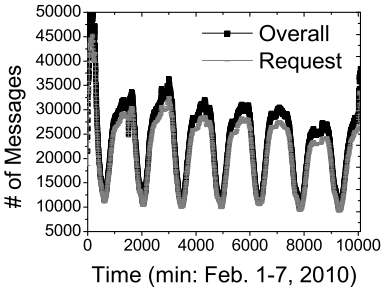


Fig. 3. # of request messages on the server

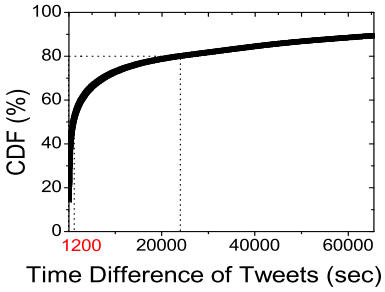
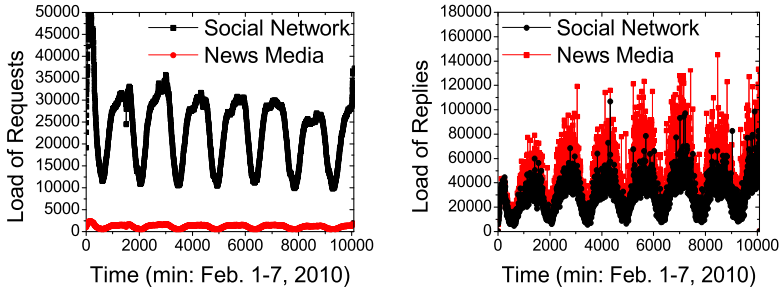


Fig. 4. Time differences of two adjacent tweets

Specially, at leisure time when users post fewer tweets, the request messages almost occupy the whole traffic. One objective of Cuckoo is thus to eliminate the unnecessary traffic caused by these polling requests. Fig. 4 is the cumulative distribution function (CDF) of the time differences between two adjacent tweets of each user. Although the burstyness of human behavior leads to tweets with small time intervals, there are still 50% of time differences larger than 1200 second and 20% larger than 24,000 second. In the worst case that polling requests are fully scheduled in these intervals, the resource waste due to unnecessary traffic is tremendous.

We further analyze the traffic load by separating it into social network usage and news media usage. The separation is based on the observations of previous studies [12], [34], [50] which report that there are two kinds of users in microblogging: social network users and news media outlets. We regard users having more than 1000 followers as *media users* and the others as *social users*. The threshold 1000 is chosen according to the homophily analysis in [34] which reports that in Twitter only users with followers 1000 or less show assortativity, one of the characteristic features of human social networks. There are 3,087,849 (99.04%) social users and 29,901 (0.96%) media users among all users. Tweets posted by media users are identified as news media usage while social users' tweets are regarded as social network usage. Fig. 5(a) shows the incoming server load in terms of received messages. Tweets posted by media users are identified as news media usage while social users' tweets are regarded as social network usage. For one request message, we calculate the percentage of media users among the requester's followees as news media usage and the rest percentage as social network usage. From Fig. 5(a), we find that the social network usage occupies the dominant proportion of incoming server load – about 95% incoming load is for social network usage while less than 5% is for news media. Fig. 5(b) reports the outgoing server load in terms of replied tweets. For each tweet within a reply message (reply to a request), we identify it into social network or news media according to whether its publisher is a media user or a



(a) Server load in terms of received messages

(b) Server load in terms of replied messages

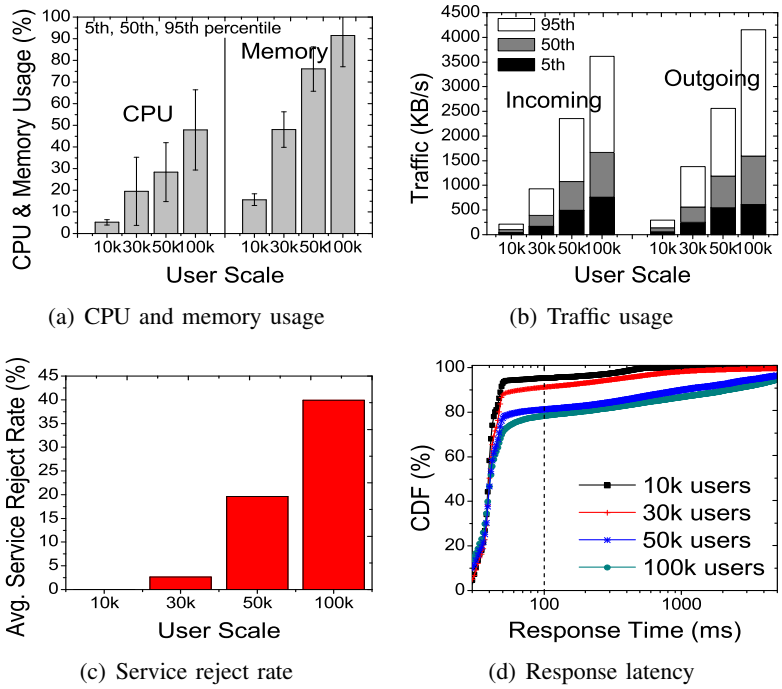
Fig. 5. Separate microblogging's server load into social network and news media

social user. We can see from Fig. 5(b) that although news media usage holds small portion of server requests (Fig. 5(a)), it occupies a great portion of outgoing server load, with 1.66 times on average more than the portion of social network usage. Thus, the dual functionality components of microblogging have distinct traffic patterns from each other, and the mix of them at same time makes the system using a single dissemination mechanism hard to scale.

C. Scalability of the Generic Centralized Microblogging System

To study the scalability of the generic centralized microblogging system, we treat Twitter as a black box and reverse engineer its operations based on Twitter traces because the details of Twitter's implementation remain proprietary. Still, we consider *post* and *request* as the main interaction models. Each user interaction is implemented through one or more connections with centralized servers. For example, to post a new tweet, a user opens a TCP connection with one server, sends the message containing the tweet, and then receives ACK to display. On the other hand, users detect new tweets by periodically polling through established connections.

We use the Twitter trace described in Section III-B to evaluate the scalability of the centralized microblogging architecture. We employ Breadth First Search (BFS) as the graph search algorithm with the start user Ustream who has over 1,500,000 followers. We prepare 4 datasets for 10,000, 30,000, 50,000, and 100,000 users respectively and prune the social links outside the datasets. We set the polling period to one minute. We run 4 server programs on a Dell PowerEdge T300, with four 2.83 Ghz quad-core 64-bit Intel Xeon CPU and 4GB of RAM. To measure CPU and memory usage of the server machine, we use the statistics provided by `vmstat` utility. For traffic usage, we use `bwm` utility to record incoming and outgoing bandwidth in every 5 seconds.



(a) CPU and memory usage (b) Traffic usage

(c) Service reject rate (d) Response latency

Fig. 6. System performance of the centralized microblogging architecture

Fig. 6 demonstrates the limited system scalability of the centralized architecture in terms of CPU and memory usage, traffic usage, service reject rate, and response time. For CPU usage, memory usage, and traffic usage (Fig. 6(a) and Fig. 6(b)), we can find the linear growth of these metrics with the increasing number of users. For example, the 50th percentile of CPU usage is 5.2%, 19.5%, 28.4%, and 47.9% for user scale 10,000, 30,000, 50,000, and 100,000 respectively. Fig. 6(c) shows service reject rate with different user scales. When user scale is 10,000, the servers can handle all the requests even at peak time. Thus, the server reject rate is almost 0 for 10,000 users. For 30,000 users, it is difficult for the servers to satisfy all the requests at peak time and the service reject rate is lower than 5%. But for 100,000 users, even in regular time, the servers are overloaded so that the reject rate is extremely high. Fig. 6(d) shows the CDF of the round-trip response time of the service with different user scales. We can see that the response latency is greatly impacted by the scale of users. When the user scale is 10,000, over 90% requests are satisfied within 100 ms. While the user scale increases to 30,000, only 80% requests have the response time less than 100 ms. For 50,000 user scale, the servers have to reject most user requests to avoid getting exhausted and keep response to limited number of user requests.

In summary, we make three key observations from our measurement studies. First, the centralized

architecture has limited scalability with the increasing number of users. Second, the main server load and traffic waste are caused by polling requests. Third, the social network and news media components of microblogging have divergent traffic patterns, which makes the system using a single dissemination mechanism hard to scale. Thus, there is a significant opportunity to eliminate the server load and traffic waste towards higher scalability by abandoning polling and decoupling the dual components.

IV. Design

In this section, we first present Cuckoo's system architecture and explain how Cuckoo is geared to the characteristics of microblogging services. Next, we describe the building blocks of Cuckoo, which put together the whole system.

A. System Architecture

1) Decoupling the Two Components: The biggest reason that microblogging systems like Twitter do not scale is because they are being used as both social networks and news media infrastructures at the same time. The two components of microblogging have very different traffic and workload patterns due to their different dissemination models. As discussed in Section III-B, although the social network component occupies more than 95% request load, the news media component holds greater portion of dissemination load, 1.66 times more than that of the social network. On one hand, the social network component is made up of most of users with limited numbers of followers. It is reported in [50] that half of Twitter users have 10 or fewer followers, 90% have less than 100 followers, and 95% have less than 186 followers. Moreover, social users do not generate much per-capita traffic. The three-week Twitter trace in [50] shows that most users sent about 100 messages during that period. On the other hand, the news media component is initiated by a small number of highly-subscribed users and then broadcasted to large numbers of other users. The study on entire Twittersphere [34] shows that there are about 0.1% users with over 10,000 followers. There are only 40 users with more than a million followers and all of them are either celebrities (e.g., Lady Gaga) or mass media (e.g., CNN). Besides, media users post tweets (named *micronews*) much more frequently than social users. The correlation analysis in [34] shows that the number of tweets grows by an order of magnitude for users who have number of followers greater than 5000. Due to the sharp gap between the dissemination models of microblogging's two components, there is no single dissemination mechanism can really address these two at the same time.

Cuckoo effectively addresses both dissemination models by decoupling the two functionality components and using complementary mechanisms. Fig. 7 shows the high-level architecture of Cuckoo, which

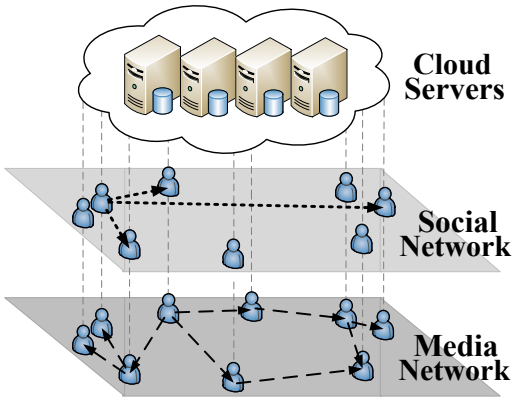


Fig. 7. Cuckoo architecture

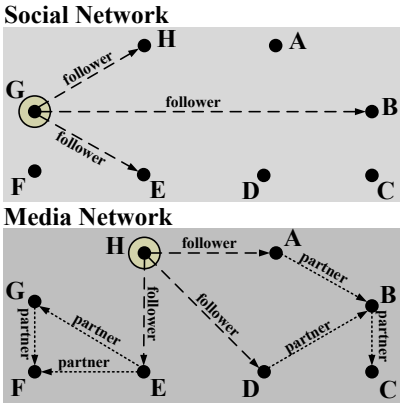


Fig. 8. Complementary content delivery

includes two kinds of logical overlay networks formed by Cuckoo peers at the network edge. For the social network component, a *social network* is formed where each publisher peer sends new tweets directly to all its follower peers in a unicast fashion. For the news media component, Cuckoo peers with similar interests form a *media network* and use gossip to disseminate micronews, i.e., enabling followers to share micronews with each other. The two overlay networks are geared to the two dissemination models of microblogging's dual components. For social users with limited number of followers, the one-to-one unicast delivery is simple and reliable. While for news media, no single peer can afford delivering micronews to large number of news subscribers. For example, in Twitter, Lady Gaga has more than 10.4 million followers. If using unicast-like delivery, it will take at least several hours to disseminate only one tweet, not to mention the overload of the sender. Thus, it is necessary to let interested peers be involved in the micronews dissemination. Fig. 8 demonstrates the two content delivery/dissemination mechanisms in Cuckoo corresponding to the two overlay networks. Besides, Cuckoo employs a set of stable peers to form a DHT (distributed hash table) that maintains all the users' connection information (e.g., IP address, port) named *node handlers* (NHs) in the distributed manner. Thus, distributed user lookup is realized: Firstly, a peer can find any other user's connection information in less than $O(log(N))$ hops on average in an N-node DHT. Secondly, we use DHT-based random walks to provide efficient partner information collection for gossip dissemination.

2) Combination of Server Cloud and Client Peers: As shown in Section III, centralized microblogging systems such as Twitter impose high server load and traffic waste, which makes centralized servers to be the performance bottleneck and central point of failure. Thus, the client-server architecture is hard to scale. Meanwhile, truly decentralized P2P systems have earned notoriety for the difficulties coping with

availability and consistency, thus achieved limited success in the past [5]. For example, FETHR [50] provides no guarantee on data delivery. A FETHR peer can receive tweets posted during its online duration while missing most tweets posted at its offline time. The follow operation will also be crippled if the potential followee is not online.

Cuckoo incorporates the advantage of both centralized and decentralized architectures by the combination of a small server base (named *server cloud*) and client peers (i.e., *Cuckoo peers*). In Cuckoo, the server cloud plays important roles including ensuring high data availability and maintaining asynchronous consistency for peers. Besides the content delivery on the two overlay networks, each Cuckoo peer also uploads its new tweets and social links to the server cloud, based on which each peer performs consistency checking at bootstrapping to detect missing events during its offline periods. By abandoning naïve polling and offloading the dissemination operation cost, the server cloud in Cuckoo gets rid of the main server load towards higher scalability. On the other hand, the servers still keep their original functions to support other operations which do not lead to performance bottleneck such as tweet searching. On the rare occasion when the server cloud is unavailable (e.g., outage [48], under attack [14]), Cuckoo peers can still find and communicate with each other. Moreover, information loss in a single location [21] can be easily recovered, since in Cuckoo both service providers and users possess the data ownership. From the service providers' perspective, Cuckoo lets them keep the same resources as in centralized systems, which is the basis of their business. In addition, Cuckoo is backward-compatible with the current polling-based web architecture, requiring no special feature on the server side.

B. Social Relation and Node State

We describe how Cuckoo peers maintain explicit and implicit social relations to form the two networks, and how "follow" is operated to build these relations.

1) Social Relation Maintenance: In typical microblogging services, a user has one or several of the following social relations: *followee*, *follower*, *friend*, and *partner*. Table 1 describes the four relations and the usage of these social information in Cuckoo. To take advantage of these social relations, each Cuckoo peer maintains four lists for friends, partners, followees and followers respectively.

To form the social network, each peer maintains the followee and follower list in its local database. The follower list is maintained according to whether the peer is a social user or a media user. The social user who has only a few followers maintains all the followers' information, while the media user with large numbers of followers maintains only a logarithmic subset. Thus, the number of entries n_i in user

TABLE I
THE DESCRIPTIONS OF SOCIAL RELATIONS AND THEIR USAGE

Social Relation	Description	Usage
Followee/Follower	Directional social links built by "follow" operations	Sending/Receiving tweets
Friend	Reciprocate social links between user pairs	Helping each other to balance load
Partner	Relations between users with common interests	Helping disseminate micronews

i's follower list can be presented as

$$n_i = max(min(F_i, H), log(F_i))$$

where F_i denotes the number of followers of user i, while H is the *partition threshold* to separate social users and media users.

To form the media network, a Cuckoo peer maintains sets of partners corresponding to each media user it follows (partners are only needed for media followees). Each Cuckoo peer collects and updates partner information using the DHT-based random walk mechanism. Note that the more people a user follows, the more information the user has to maintain so as to join multiple dissemination processes, which to some extent suppresses the behavior of evangelists (e.g., spammers or stalkers) [32].

Friend is a reciprocate social link which indicates that two users may be *acquaints* with each other and willing to help each other [32]. Each Cuckoo peer keeps connections with G friend peers. The peer and its friend peers make up the *virtual node* (VN) via request redirection, *i.e.,* friends help each other to balance load and improve availability. For example, we assume that the VN of user A consists of the Cuckoo peer A and its two friend peers B and C. When peer A is leaving the system, it asks one friend peer (B or C) to fill the vacant as if itself is still there (the successive friend peer has all the information of user A).

2) Follow: The "follow" operations explicitly build the followee-follower relations between user pairs, which forms the basis of the social network. To follow a specific user, a Cuckoo peer first lookups the followee's NH according to his (or her) `userId`[6] via the DHT. The DHT maintains all users' NHs in the key-value format (`userId` as key and NH as value). Then, the peer sends a follow request that attaches its profile to the followee peer using the NH. There are 2 cases according to whether the followee peer is online or not: (1) If the followee peer is online, it receives the request and sends back a reply directly

[6]In Cuckoo, each user is assigned a unique `userId` by the server cloud at registration, which simplifies the authentication and Id assignment.

to the requester. After receiving the reply, the follower sends a notification to the server cloud to inform the built relation; (2) If the followee peer is offline, the requester submits its willing to the cloud. The cloud checks the validity and replies the results. Each Cuckoo peer checks the inconsistency between the follower list maintained locally and the one maintained by the server cloud at online bootstrapping. If there exist some new followers during its offline period, the peer sends replies as compensation. Note that the consistency checking does not require complete comparison of the two follower lists. As long as the server cloud maintains users' follower list in reverse chronological timeline like Twitter, the Cuckoo peer is able to send the cloud its last recorded follower's `userId` and get back the new guys.

3) Content Query: Cuckoo also allows users to "query" information of any other user like what Twitter provides. The query process is similar as the follow process described above, *i.e.,* the query user first lookups the target user's NH (in the case the query user has no information about the target user), while the user being queried sends back the reply containing the queried contents. Cuckoo supports different query parameters. For example, query for a user's tweets posted between time t_1 to t_2, for a tweet with specific `statusId`[7], etc. In the worse case that the target user is not online, the query peer can still fetch the required contents from the omnipotent server cloud.

C. Unicast Delivery for the Social Network

When a user posts a new tweet, the microblogging service should guarantee that all the user's followers could receive that tweet. For the social network where users' social links are limit in size (e.g., a few hundred followers), serial unicast-like content delivery is simple and reliable. The publisher peer tries to push the newly posted tweet via direct unicast socket to each follower. This is achieved by locally caching each follower's latest node handler (NH). To ensure that followee peers always keep the up-to-date NHs, a user informs all his (or her) followees when changing the NH, e.g., in the case that the user accesses the service using different devices in different places. Moreover, the user is required to update the NH replicas in the DHT so that any other user is able to search up-to-date NHs via the DHT.

The unicast-like delivery for the social network can ensure all the online followers to receive their followees' new updates in time. However, for offline followers absent from the delivery process, they should regain the missing tweets when re-entering the system. Cuckoo achieves this also by consistency checking, i.e., each peer fetches the bunch of tweets posted at its offline period from the server cloud at bootstrapping. Since tweets are maintained in the reverse chronological timeline, a new coming user's

[7]In Cuckoo, each tweet is bounded to a unique `statusId`.

missing parts can be efficiently detected by giving his (or her) last departure time or the `statusId`[8] of his (or her) last received tweet. This checking process is also applicable for media users in the media network. Note that the consistency checking is only used to detect missing tweets and social links, not to check the NHs.

D. Gossip Dissemination for the Media Network

In the media network, media users (e.g., CNN, Lady Gaga) cannot afford sending updates to all their followers. In this case, Cuckoo uses gossip-based dissemination, i.e., enable interested users involved in the micronews dissemination process. Gossip (also known as Epidemic, Rumor) information dissemination has been proved to be scalable and resilient to network dynamics. The theoretical support provided in [31] proves if there are N nodes and each node gossips to $log(N) + c$ other nodes on average, the probability that everyone gets the message converges to $e^{-e^{-c}}$, very close to 1.0 without considering the bandwidth constraint, latency, failure, etc. This result provides a guideline for Cuckoo's partner management, i.e., maintain the number of partners (called *fanout*) to be logarithmic of the number of followers.

Cuckoo adopts the simple "infect and die" model [20] for micronews dissemination. Peers, once infected, remain infectious for one round precisely, before dying, i.e., the peer followed a media user gossips each of the followee's tweet (i.e., micronews) exactly once, namely after receiving that tweet for the first time, but will not further gossip even when receiving subsequent copies of the same tweet. Initially, the media user sends a gossip message containing the micronews to a subset of its online followers. Upon receiving the gossiped message, the Cuckoo peer determines whether it has received this micronews or not by checking the `statusId`. For a new micronews, the peer saves it locally and continues gossiping to the $log(n) + c$ partners, where n is the number of all the online followers of the media user. Otherwise, the peer discards the message and takes no action. In this case, the number of rounds R, i.e., network hops necessary to disseminate a micronews to all the follower nodes respects the following equation [20]:

$$R = \frac{log(n)}{log(log(n))} + O(1)$$

which shows that it takes at most a logarithmic number of steps for a micronews to reach every online follower.

An alternative for micronews dissemination is to use the application-level multicast tree. However, the multicast tree is vulnerable to peer churn and its maintenance is complicated, which makes it impractical

[8]In Cuckoo, each tweet is bounded to a unique `statusId`.

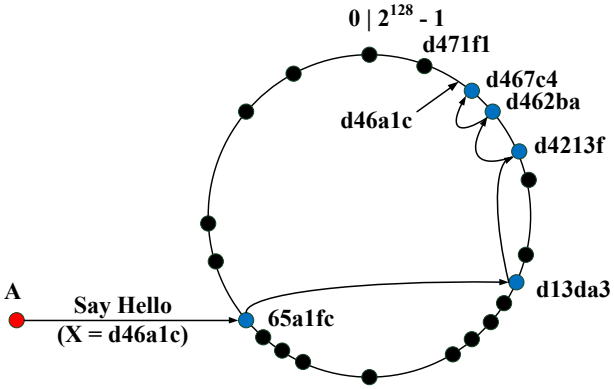


Fig. 9. An example of "say hello" announcement

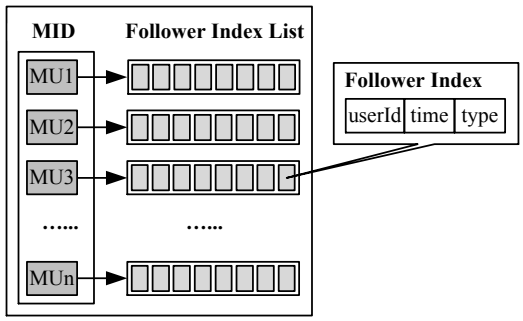


Fig. 10. The format of media user indices maintained on DHT nodes

to dynamic microblogging user access.

Partner Collection Mechanism. To discovery online partners in case of malfunctions (e.g., churn), we design a DHT-based partner collection mechanism. The mechanism is made up of two steps: *announcement* and *discovery*. For announcement, a joining peer picks a random `nodeId` X and asks a bootstrap node to route a special *hello message* on the DHT using X as the destination key. The hello message announces the new peer's presence as well as its interests on media users (i.e., the `userIds` of its media followees). This hello message is routed to the DHT node with `nodeId` numerically closest to X. The nodes along the DHT route overhear the message and check the new node's interests. In this way, the stable nodes in the DHT construct probabilistic follower indices of media users (named MUI, media user indices). The online peers says hello periodically via the DHT to keep freshness. We set the period as 10 minutes in Cuckoo. Fig. 9 gives an example of the "say hello" announcement. In this figure, the new coming Cuckoo peer A routes a hello message which attaches her `userId` and media followee list to $X = d46a1c$. During this process, the five stable nodes ($65a1fc$, $d13da3$, $d4213f$, $d462ba$, and $d467c4$) along the DHT route overhear this passing-by hello message and update their MUI. Each stable node on the DHT maintains a set of MUI. Fig. 10 shows the format of MUI, where MID is the media user's `userId` and the follower index list is maintained as a LRU queue. Each follower index (FI) contains three important attributes: the follower's *userId*, the last update *time*, and the follower's Cuckoo client *type* (introduced in Section IV-E).

To discover new online partners of a media user, a Cuckoo peer uses DHT-based random walks (e.g., [10], [11], [43]) to collect partners over the DHT topology by checking these indices. To look up

m new partners of the media followee with `userId` F, node A generates a partner query and sends it to all the nodes in its in its DHT routing table. The query message is tagged with F, m, an empty array a and the routing table row r of node A. When a node receives a partner query, it checks the MUI it is maintaining with F and appends the follower indices with MID equal to F into a. Then, the node forwards the partner query to all nodes in its DHT routing table in rows greater than r if any. This process is complete until the length of a equals to m. Note that the process described above is a flooding-based partner discovery. The random walks can be achieved by performing a breadth-first traversal of the flooding tree, i.e., in each round selecting a random subset of nodes among all the nodes in DHT routing table. The DHT-based flooding/random walk ensures that nodes in the DHT are visited only once during a collection process [10]. Since partner collection is only needed for media users who have high popularity (i.e., their follower number occupies a large portion of user population), the random walk-based collection is efficient[9].

E. Support for Client Heterogeneity

User clients in deployed microblogging systems are heterogenous with different bandwidth, energy, storage, processing capacity, etc. For example, the CEO of Twitter recently stated that over 40% of all tweets were from mobile devices, up from only 25% a year ago [39]. Thus, it is important to support client heterogeneity in Cuckoo, in consideration of the economic burden of increased load on mobile peers such as higher cost for network usage (due to expensive or limited mobile data plans), as well as higher energy consumption resulting in reduced battery life.

Cuckoo differentiates user clients into three categories named *Cuckoo-Comp*, *Cuckoo-Lite*, and *Cuckoo-Mobile*. Cuckoo-Comp is designed for *stable nodes* which reside in the system for a relatively long time (more than 6 hours per day in our experiments). These stable Cuckoo-Comp peers construct the DHT to support user lookup and partner collection. The stable nodes are only a small subset of all the nodes (about 15% in our dataset), but their relatively long life spans allow them to keep the DHT stable with low churn. Several mechanisms [4], [58] can be integrated into Cuckoo to identify stable nodes in overlay networks, e.g., the nodes already with higher ages tend to stay longer. Cuckoo-Lite is designed for lightweight clients (e.g., laptops with wireless access) while Cuckoo-Mobile is for mobile devices (e.g., smart phones). Neither of them joins the DHT overlay and the main difference between them is that Cuckoo-Mobile peers do not participate in the gossip dissemination process in the media network

[9]It is known that the flooding/random walk-based search is efficient for locating popular (well replicated) items in P2P networks [35], [61].

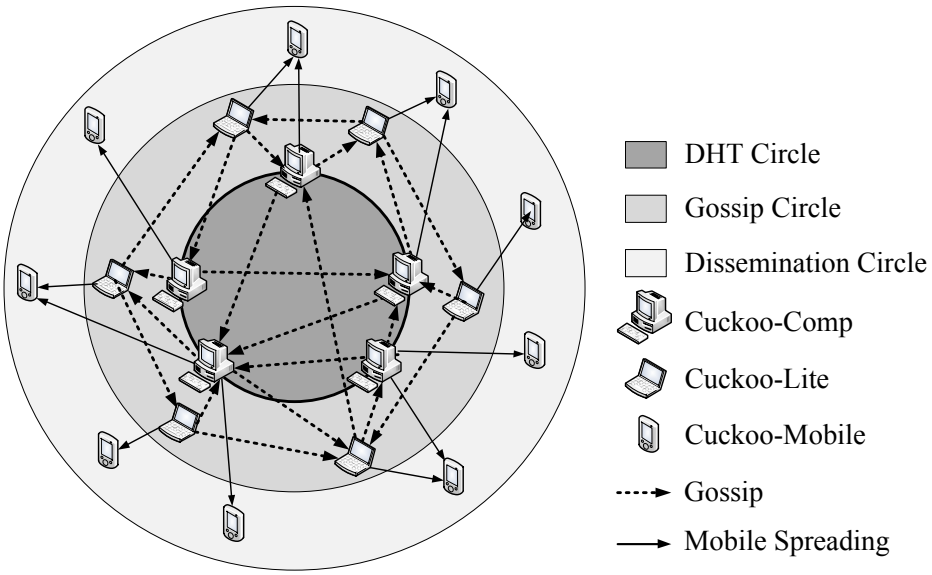


Fig. 11. Content dissemination with heterogenous user clients

while the Cuckoo-Lite peers do (as the Cuckoo-Comp peers). Since mobile devices have energy and bandwidth constraints, they have no incentive to further forward the received messages. Thus, we regard the Cuckoo-Mobile peers as *leaf nodes*. In addition, We call both Cuckoo-Comp and Cuckoo-Lite peers as *gossip nodes* since they have no difference in the gossip process. For gossip nodes, heterogeneity-aware gossip (e.g., [24], [30]) can be used to tune the fanout and *fanin* (the number of times a node is chosen as a gossip target). For instance, using heterogeneous gossip [24], each gossip peer tunes its fanout as $f = ((log(N) + c) \times b)/\bar{b}$, where b is its own capacity and $\bar{b}$ is the estimate of the average upload capacity of all the network nodes. The dissemination is initiated from the publisher, gossiped through the Cuckoo-Comp and Cuckoo-Lite peers (gossip nodes), and simultaneously spread to Cuckoo-Mobile peers (leaf nodes). Fig. 11 illustrates the content dissemination mechanisms that support user heterogeneity. In Fig 11, each gossip node sends the newly received tweet to two other gossip nodes, while spreading the tweet to one leaf node.

The mobile spreading is realized by the partner collection mechanism which is introduced in Section IV-D. Since all types of Cuckoo clients (no matter gossip nodes or leaf nodes) announce their hello messages to the DHT overlay. Each follower index (FI) in MUI records the type of Cuckoo clients (see Fig. 10). Besides the array a to record m gossip partners, each partner query message includes another array a' for mobile partners. Using DHT-based random walks, when a query message meets a partner P, it checks

the Cuckoo type of node P in the FI. If node P is a gossip node, the query message appends P into array a and continues finding the rest $m-1$ partners (i.e., the process described in Section IV-D). Otherwise node P is a leaf node, the query message puts P into array a' and still ties to find m partners, i.e., collecting mobile partners in a piggyback manner. In this way, the gossip nodes also maintains extra Cuckoo-Mobile peers for mobile spreading. For the partner query with the length of array a equal to m, the expectation of the length a' is $(\frac{\rho}{1-\rho}) \times m$, where ρ is the percentage of Cuckoo-Mobile peers over all the Cuckoo peers.

Note that the support for client heterogeneity in Cuckoo only makes sense in the media network. For the social network, all kinds of Cuckoo peers using the same unicast delivery mechanism introduced in Section IV-C.

F. Message Loss Detection and Security Issues

1) Detecting Lost Tweets: While Cuckoo's probabilistic dissemination (e.g., gossip process) achieves high resilience to node failures as well as high coverage rate, it provides no guarantee that each tweet is reached to all the nodes due to the intrinsic uncertainty brought by the randomness. Thus, we need a mechanism to detect which tweet fails to arrive. Cuckoo exploits the `statusId` to solve this problem. In Cuckoo, each tweet is bounded to a unique `statusId`. The `statusId` is made up of two parts. The prefix is the `userId` of its publisher and the postfix is a `long` sequence number maintained by the publisher's counter. By checking `statusIds` of received tweets, a Cuckoo peer can easily identify gaps between the sequence numbers of `statusIds`. Then, the Cuckoo peer could fetch the lost tweets from either other peers by content query (Section IV-B.3) or the server cloud if the peers being queried are not online.

2) Security: A number of systems and architectures are proposed to safeguard sensitive user data against untrusted third parties as well as malicious attackers [7], [15], [51]. While the main concern of this paper is not on the security aspect, Cuckoo can easily integrate the security components to protect microblogging users from different attacks.

One important concern is authentication since some malware may impersonate normal users to distribute spam. To defend against the forged tweets, digital signature based on asymmetric key cryptography can be employed, *i.e.,* publishers generate and attach digital signatures with the tweets using their private keys, while followers verifying the authenticity of the received contents using the followee's public key. Similarly, forward and gossip peers are asked to attach the digital signatures of the original tweets for

authenticity checking. A Cuckoo client generates the public/private key pair at the first time it is launched, and then uploads the public key to the server cloud. A follower can obtain the followers' public keys in either the "follow" process (Section IV-B) or from the server cloud. Key revocation requires re-keying, using the similar process as message delivery/dissemination to distribute the new public key. Still, in any case, peers can update for the newest key from the server cloud.

In addition, the centralized control of user data on the server cloud raises the risk of privacy violation. For example, service providers may generate revenue by disclosing users' information to third-party advertisers and websites without explicit consent from users. In this case, the social users who refuse to make their tweets publicly access encrypt contents before uploading to the server cloud. To preserve privacy, they do not upload their public keys onto the server cloud but only directly share keys with specified users.

Compared with centralized systems, Cuckoo has better resistance to malicious attacks that target on crippling the centralized server, *e.g.,* DoS attacks and content censorship. Even though DoS attacks may make the server cloud unavailable [14], they can hardly impact the information sharing between peers. Cuckoo peers are still able to discover each other and deliver/disseminate messages via the social and media networks. Similarly, content censorship by governments on the server cloud cannot restrain direct information exchanging between peers.

For the brute-force attacks that malicious nodes generate unwanted traffic (*e.g.,* replay, spam) to harass normal peers' operations in the distributed social and media networks, trust and reputation mechanisms can be employed. The social relations maintained on each peer (Section IV-B.1) provide nature trust relationships to build the reputation model, based on which unwanted communications can be thwarted [38].

V. Experimental Evaluation

To evaluate Cuckoo's performance, we run Cuckoo prototype using trace-driven emulation of the Twitter trace containing 30,000 Twitter users which reconstructs the part of Twitter's traffic patterns from Feb. 1 to Feb. 7, 2010. We evaluate Cuckoo in two aspects: the performance gain of the server cloud under Cuckoo architecture, as well as the performance of message sharing and micronews dissemination between Cuckoo peers.

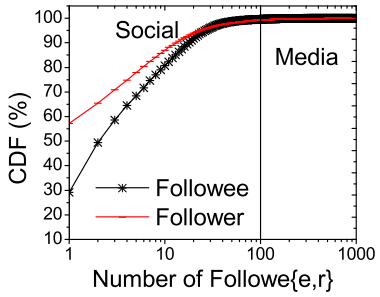


Fig. 12. Number of followe{e,r}s

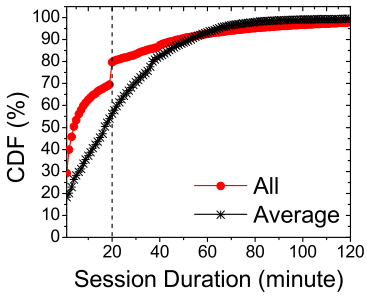


Fig. 13. User session duration

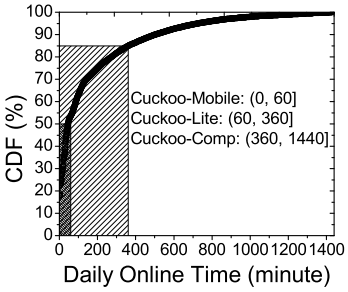


Fig. 14. User daily online time

A. *Dataset and Processing*

We use the dataset described in Section III-B to evaluate the system performance. The raw dataset contains 3,117,750 users' profiles, social relations, and all the tweets maintained on the Twitter site[10] from Mar. 6 to Apr. 2, 2010. To allow fair comparison, we focus on user access patterns during the one-week period from Feb. 1 to Feb. 7, 2010 as discussed in Section III.

For each user, we distill his (or her) tweets posted from Feb. 1 to Feb. 7, 2010. We throw the inactive users who posted nothing (have no activity) in this period since these users make no impact on the system. We use BFS as the graph search algorithm with the start user Ustream to create the dataset containing 30,000 user information to match the capacity of the emulation testbed. For the social links, we prune the irrelevant users outside the dataset. Fig. 12 plots the CDF of number of users' followees and followers in the dataset. We set $H = 100$ as the partition threshold to separate social users and media users, i.e., the users with more than 100 followers are considered as media users while social users have less than 100 followers. There are 29,874 (99.58%) social users and 126 (0.42%) media users in the experiment dataset. Note that we use $H = 100$ instead of 1000 in Section III-B due to the social link pruning. We can see from Fig. 12 that less than 5% users are media users.

To emulate Twitter users' behavior, we need users' online time, *i.e.,* the session durations of each user. However, the session durations cannot be obtained by crawling the Twitter site (Twitter does not indicate whether a user is online or not). As the best of our knowledge, there is no session duration dataset available for microblogging services. So we choose the OSN session duration dataset provided by [27]. The dataset contains 1705 MySpace users' one-month session duration log. To meet the one-month log, we first pick out all the users' tweets posted in February, 2010. Then for each Twitter user with t number of tweets, we select the duration log containing s_i number of sessions, satisfying

[10]Twitter only reserves about 3000 tweets per user at most and discards the previous user tweets.

$\min\{s_i \geq t\}$ $(i = 1, 2, \cdots, 1705)$. In other word, each tweet is bounded to one session duration. We set one tweet to be posted in the period of one session and merge overlapped sessions to be one longer session. Thus, each user maintains an `eventLog` containing the timestamps of three events: `Start_Session`, `Post_Tweet` and `End_Session`. Each user has a timetable specifying the timestamps of 3 events. Fig. 13 plots the CDF of the average session duration of one user and the CDF of all the session durations. We classify the 3 types of Cuckoo users according to their total online time in the one-week period, i.e., Cuckoo-Comp users are those whose daily online time exceed 360 minutes, and Cuckoo-Mobile users spend less than 60 minutes online per day, and the others are Cuckoo-Lite users. Fig. 14 shows the CDF of the total online time and the classification. Note that this session duration dataset gives a pessimistic estimation for Cuckoo users' online behavior. As a centralized OSN service, MySpace operators employ *session timeout* set as 20 minutes to prevent user polling so as to mitigate server load (see the vertical jump in Fig. 13). A user session is suppose to be disconnected as long as the user has no interaction request (e.g., post tweets, follow friends) to the server in 20 minutes. However, the situation is completely opposite in Cuckoo. Cuckoo encourages users to stay online as long as possible. The duration of users' online time makes no impact on the server cloud because pollings are abandoned in Cuckoo. The more Cuckoo-Comp and Cuckoo-Lite users in the system, the better performance can be expected. We mark the design of incentive mechanisms for Cuckoo peers as our future work.

B. Implementation and Deployment

We have built a prototype of Cuckoo using Java. Our implementation comprises both the Cuckoo peer and the server cloud. The prototype of Cuckoo peer adds up to 5000 lines of Java code including the three types of clients, i.e., Cuckoo-Comp, Cuckoo-Lite, and Cuckoo-Mobile, with different software components. We use Java socket to implement end-to-end message delivery. We design and implement different types of application layer messages in Cuckoo. For example, `DirectUpdateMsg` is used for unicast delivery in the social network while `GossipUpdateMsg` for gossip dissemination in the media network. Different types of messages are identified by the `type` attribute. For local data management, we use XML to store node states and social relations including followee/follower profiles, followee tweets, and self-posted tweets. We use `JDOM` for constructing new entries and updating elements. However, since the high memory and CPU cost of the DOM model, we choose `StAX` for reading and lookup XML entries using the streaming pull parsing model. We choose Pastry [46], PAST [47] and their implementation FreePastry as our overlay infrastructure for Pastry's good properties (*e.g.,* locality awareness) as well as FreePastry's platform independence (Java source code). Note that Cuckoo does not rely on any Pastry's

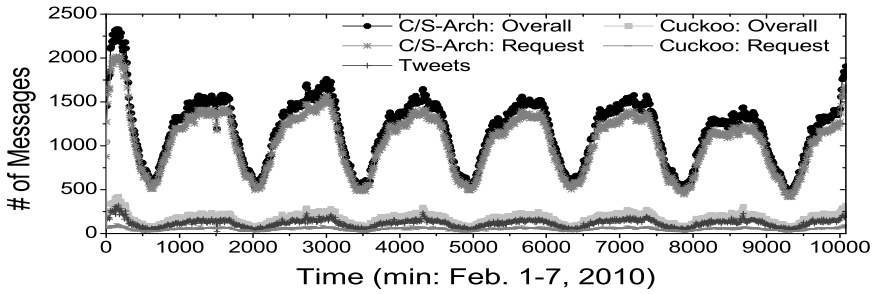


Fig. 15. Message overhead of the server cloud.

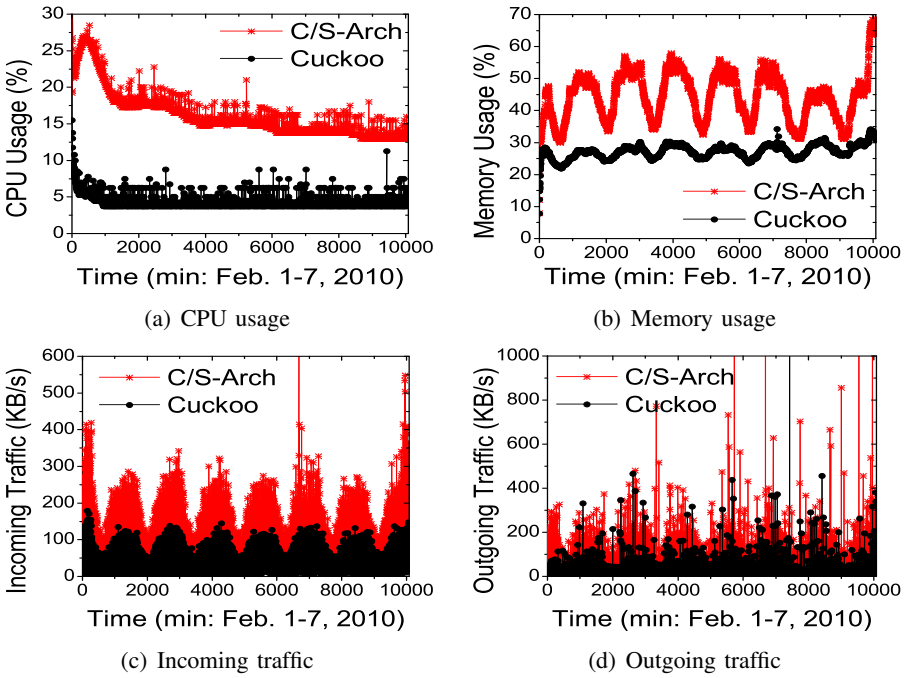
special feature (*e.g.,* leaf set), and thus it is applicable for any structured overlay that supports the key-based routing (KBR) API defined in [16]. The server cloud prototype adds up to 1500 lines of Java code. It uses plain text files to store all users' information.

We have deployed 30,000 Cuckoo peers without GUI on 12 machines including 3 Dell PowerEdge T300, each with four 2.83 Ghz quad-core 64-bit Intel Xeon CPU and 4 GB of RAM, and 9 Dell Optiplex 755, each with two 3.00 Ghz Intel Core 2 Duo CPU and 3.6 GB of RAM. We have deployed 4 servers to build the server cloud on another Dell PowerEdge T300 machine and let them share storage, so that these servers have no caching inconsistency problem. All these machines are installed Ubuntu 9.10 Server Edition. We located these machines into 2 LANs connected by a 10 Gb/s Ethernet cable. We ran the Cuckoo peers based on the one-week Twitter trace described in Section V-A. Still, we use `vmstat` utility to measure CPU usage and memory usage for the cloud machine and `bwm` utility to record server bandwidth in every 5 seconds.

C. Server Cloud Performance

In order to characterize the performance gain of Cuckoo, we compare the performance of the server cloud under the Cuckoo architecture with that under the traditional client-server architecture (Section III-C), denoted as "C/S-Arch". Remember that Cuckoo is fully compatible with the current polling-based web architecture, i.e., Cuckoo does not require any extra functionality on the server side. Thus, the server cloud implementations for Cuckoo and C/S-Arch are exactly the same. For fair comparison, both of them use the same machine and system configuration.

1) Message Overhead: We record all the messages received by the server cloud. Fig. 15 shows the number of different types of messages in time series received by the server cloud of Cuckoo and C/S-Arch. We can see that the overall number of messages received by Cuckoo's server cloud is far less than C/S-Arch's. For 30,000 users, the server cloud in C/S-Arch receive over 1500 messages per minute at



(a) CPU usage (b) Memory usage

(c) Incoming traffic (d) Outgoing traffic

Fig. 16. Resource usage of the server cloud

peak time and over 500 messages per minute at leisure time, while in Cuckoo even at peak time, the message overhead is only about 300 messages per minute – 20% of that in C/S-Arch.

Moreover, we observe that Cuckoo's message overhead is mainly made up of tweets, i.e., posting tweets onto the server cloud, which is essential for service providers. Request messages only occupy about 25% percentage of the overall message overhead in Cuckoo. However, in C/S-Arch, the blind polling requests lead to over 90% percentage of the overall message overhead, causing large traffic waste. Besides, the server cloud of C/S-Arch experience much more severe fluctuation of message overhead. According to Fig. 15, the peak-valley difference experienced by the C/S-Arch's server cloud is about 1600-500=1100 messages, while for Cuckoo, it is only about $220-80=140$. In summary, Cuckoo can save 80% message overhead on the server cloud compared with C/S-Arch.

2) Resource Usage: We characterize the resource usage of the server cloud in Cuckoo compared with that in C/S-Arch in terms of CPU usage, memory usage, as well as incoming and outgoing bandwidth usage.

Fig. 16(a) shows the server cloud's CPU usage of Cuckoo and C/S-Arch in time series. We can see that Cuckoo achieves notable reduction of CPU usage compared with C/S-Arch – the server cloud in Cuckoo consumes 60% less CPU than C/S-Arch, with the average value being 5%. The main usage of

CPU is for the database lookup process and I/O scheduling. Since the server cloud in Cuckoo receives far less requests than that in C/S-Arch, the CPU usage reduction is not surprising.

Fig. 16(b) shows the server cloud's memory usage of Cuckoo and C/S-Arch in time series. Compared with CPU, memory usage is more sensitive to the number of requests. In other words, memory usage is significantly impacted by the message overhead. As a result of message overhead saving (see Fig. 15), Cuckoo effectively reduces the memory usage compared with C/S-Arch. The server cloud of C/S-Arch consumes 50% of memory at peak time and 30% at leisure time, while the Cuckoo cloud's memory usage is around 25%. In summary, Cuckoo achieves about 50%/16% memory usage reduction for the server cloud at peak/leisure time.

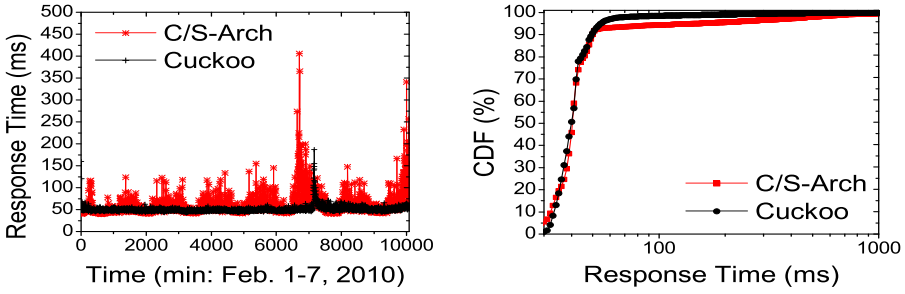
Fig. 16(c) and Fig. 16(d) demonstrate the server cloud's incoming and outgoing bandwidth usage of Cuckoo and C/S-Arch in time series. The bandwidth usage is directly decided by the message overhead. The larger volume of messages the server cloud receives/sends, the more bandwidth is consumed. Cuckoo effectively reduces the incoming and outgoing bandwidth consumed, with about 120 KB/s, 70 KB/s at peak, leisure time for incoming bandwidth, about 200 KB/s, 90 KB/s at peak, leisure time for outgoing bandwidth. The incoming, outgoing bandwidth consumed for C/S-Arch is 300 KB/s, 400 KB/s at peak time and 400KB/s, 100 KB/s at leisure time, respectively. Thus, the server cloud in Cuckoo saves about 50% bandwidth consumed for both incoming and outgoing traffic compared with C/S-Arch.

3) Response Latency: We examine the response latency provided by the server cloud of Cuckoo and C/S-Arch, and the results are shown in Fig. 17. Fig. 17(a) shows the response latency in time series and Fig. 17(b) shows the CDF of all the recorded response latency. We can see that at leisure time, the response latency of Cuckoo and C/S-Arch is similar (about 50 ms). However, at peak time, the response latency of Cuckoo is far less than that of C/S-Arch. The response latency of Cuckoo is relatively smooth, being around 50 ms in most time, while at peak time the response latency of C/S-Arch is more fluctuant and higher that can reach 100 ms or more.

Since in Cuckoo the peak-valley difference is smaller than that of C/S-Arch in terms of CPU, memory usage as well as bandwidth consumed, even at peak time the server cloud has enough resources to satisfy all the requests and posts. In contrast, at peak time the C/S-Arch server cloud has too much burden so that it cannot handle all the concurrent requests and posts normally at the same time.

D. Cuckoo Peer Performance

In this section, we characterize the performance of Cuckoo peers. Each Cuckoo peer maintains a message log that records all the received, sent, and forwarded messages. By collecting the message logs

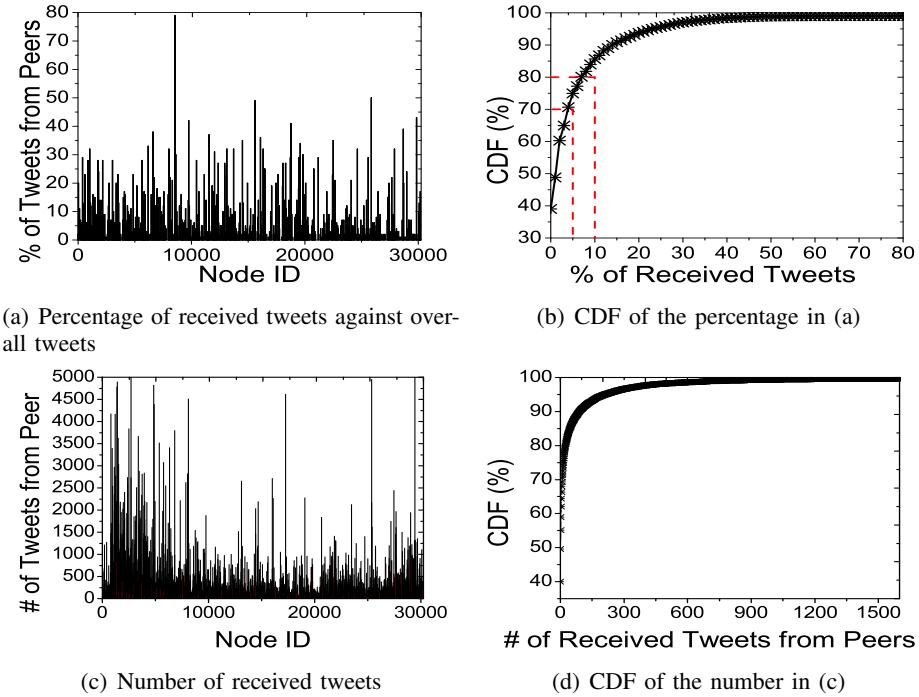


(a) Response latency in time series (b) CDF of response latency

Fig. 17. Response latency of the server cloud

from all the Cuckoo peers, we analyze the performance of message sharing. Moreover, the message logs specify the detailed information of gossiped messages including `statusIds` of gossiped tweets, gossip hops, and redundancy. Based on these, we analyze the performance of gossip dissemination.

1) Message Sharing: Fig. 18 shows the performance of message sharing between Cuckoo peers. Fig. 18(a) shows for each peer, the percentage of tweets received from other Cuckoo peers other than from the server cloud, while Fig. 18(b) is the CDF of the percentages. According to Fig. 18(a) and 18(b), around 30% users get more than 5% of their overall subscribe tweets from other peers, and around 20% get more than 10%. On the other hand, Fig. 18(c) and 18(d) give another quantitative measure, i.e., the number of received tweets from other peers. The performance of message sharing is mainly decided by two aspects: users' access behavior and users' online durations. Generally speaking, the more overlapped behavior the followee-follower pairs have, the higher probability that follower peers could receive tweets from followees. For online durations, the longer the user stay online, the higher probability he (or she) can receive tweets from other peers. In the extreme case that a user keeps online all the time, he (or she) cannot miss any subscribed tweet without fetching from the server cloud. We should note that the duration dataset used in our experiments leads to a pessimistic deviation of the message sharing performance. Due to the centralized architecture of existing OSN services, the OSN operators employ *session timeout* to reduce users' polling so as to mitigate server load. In our duration dataset, the timeout is set as 20 minutes (see the vertical jump in Fig. 13) [27], i.e., a user session is supposed to be disconnected as long as the user has no interaction (e.g., post tweets, follow friends) with the server in 20 minutes. However, the situation is completely opposite in Cuckoo where users are highly encouraged to stay online as long as they can. The long online durations of users can significantly improve the performance of message sharing without performance degradation of the server cloud (no polling any more). Thus, we can expect better performance of message sharing in Cuckoo.



(a) Percentage of received tweets against overall tweets

(b) CDF of the percentage in (a)

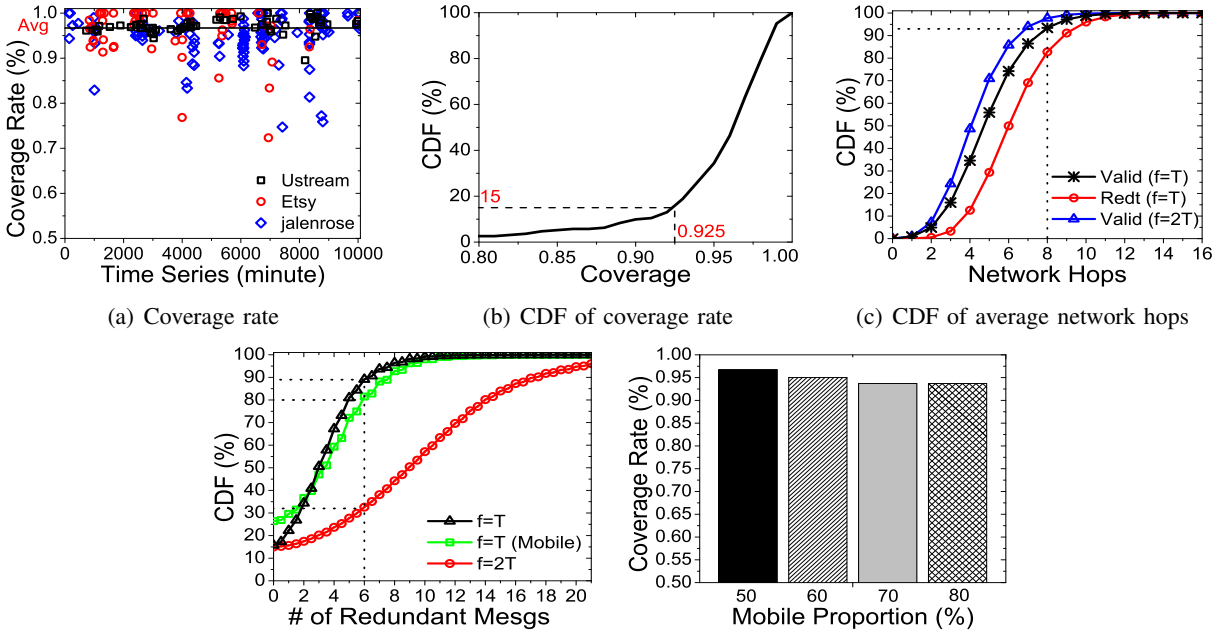
(c) Number of received tweets

(d) CDF of the number in (c)

Fig. 18. Performance of message sharing

2) Gossip Dissemination: We evaluate the performance of micronews dissemination in the media network. Media users who have more than 100 followers use gossip to disseminate tweets, i.e., *micronews* (see Fig. V-A(a)). In our experiment, each Cuckoo-Comp or Cuckoo-Lite peer (i.e., gossip node) maintains the fanout $f = T$ for one media followee, where $T = log(n) + 2$. In addition, due to the mobile spreading mechanism that delivers tweets to leaf nodes, a gossip peer is likely to maintain some Cuckoo-Mobile partners. Since leaf nodes occupy 50% of all the nodes in our dataset (Fig. 14), a gossip peer maintains at most $2T$ partners for gossip and mobile spreading. For instance, the media user Ustream with 29,927 followers sends no more than 24 messages for each dissemination process. Fig. 19 demonstrates the performance and overhead of Cuckoo's micronews dissemination in terms of coverage, hops, and redundancy. The coverage and redundancy are conflict with each other impacted by the fanout. Larger fanout leads to higher coverage while imposing higher redundancy. Thus, the fanout should be chosen carefully to tradeoff the two metrics.

For the ease of presentation, we select three typical media users to illustrate the dissemination performance in terms of coverage. Fig. 19(a) shows the coverage rate of each dissemination process in time series. In this figure, user Ustream, Etsy, jalenrose with 29,927, 6,914, 696 followers publishes 61, 59, 117 tweets respectively in the one-week period. Fig. 19(b) shows the CDF of the coverage rate. We can



(a) Coverage rate (b) CDF of coverage rate (c) CDF of average network hops

(d) CDF of average redundant messages (e) Coverage with different mobile node portion

Fig. 19. Micronews dissemination

see from the CDF that around 85% dissemination processes cover over 92.5% of users among all the online followers of the media users, with the average coverage rate equal to 96.7%. All the dissemination processes of the three media users have coverage rate higher than 70%, and there are only few process with coverage rate lower than 80% due to the uncertainty of gossip and user asynchronism.

Fig. 19(c) shows the CDF of the network hops of tweets received by all the users. We compare the hops of *valid* tweets (i.e., tweets received for first time) with those of *redundant* ("Redt" in Fig. 19(c)) tweets as well as the hops of valid tweets with doubled fanout, i.e., $f = 2T$. We can see that 90% of valid micronews received are within 8 network hops, while redundant tweets use more hops to reach the followers. On the other hand, increasing fanout reduces limited network hops of dissemination. With $f = 2T$, each user only reduces less than one hop on average to receive micronews, while the partner maintenance overhead is doubled.

We further study the performance of Cuckoo's client heterogeneity support by tuning the proportion of Cuckoo-Mobile peers among all the nodes. Fig. 19(e) shows the average coverage rate of micronews dissemination with different mobile proportion. According to the figure, Cuckoo achieves stable dissemination performance in terms of coverage under client heterogeneity. Even when the mobile proportion reaches 80%, the dissemination can still achieve over 90% coverage. Nevertheless, when the mobile

proportion is high, the high dissemination coverage is based on the high overhead on Cuckoo's gossip nodes: each gossip node is likely to maintain and spread micronews to extra $(\frac{\rho}{1-\rho}) \times T$ leaf nodes, where ρ is the proportion of Cuckoo-Mobile peers (Section IV-E).

3) Client Overhead.: In Cuckoo, the client overhead for message delivery/dissemination is twofold: outgoing traffic overhead and incoming traffic overhead. The outgoing traffic overhead is bounded according to the delivery/dissemination mechanisms. The overhead of unicast delivery is n-tweet sending per process, where n is the number of online followers. For gossip dissemination, the overhead is f-tweet sending per process where f is the fanout. The incoming traffic overhead is mainly caused by receiving redundant messages. Fig. 19(d) shows the CDF of average redundant messages received by each peer for one dissemination process, compared with that of $f = 2T$. Moreover, we pick out the mobile users (i.e., Cuckoo-Mobile) and show their redundancy. According to the figure, around 89% of users receive less than 6 redundant tweets for one dissemination process while 80% of mobile users among them receive less than 6, which is of acceptable overhead due to the small size of tweet messages. On the other hand, the increase of fanout causes larger redundancy. For $f = 2T$, more than 65% of users receive over 6 redundant tweets, while the increase of coverage rate is trivial (less than 1%). Other client overhead includes the overhead for DHT maintenance (only for Cuckoo-Comp peers) and partner maintenance (e.g, announcement, discovery).

VI. Discussion and Future Work

We discuss some current missing components of Cuckoo, which also shed light on directions for our future work.

A. Support for Clients behind NATs

Network Address Translation (NAT) causes well-known difficulties for communication among peers without globally valid IP address. Several NAT traversal techniques have been proposed including relaying [36], connection reversal, and hole punching [45], [53]. Most of these NAT traversal techniques can be used in Cuckoo to enable two peers to set up a peer-to-peer session, so as to send unicasts to systems behind a NAT. Take hole punching as an example, the Cuckoo peer A behind a NAT tries to record and store (e.g., by relaying) its *two* endpoints (i.e., the *public* endpoint and the *private* endpoint) in its corresponding node handler into the DHT. Since Cuckoo peer B can obtain the two endpoints of A from the DHT, peer B can establish a direct TCP/UDP session with A using the hole punching technique in [53]. However, this technique needs assistance from *rendezvous servers* at publicly routable IP to help

set up the direct connection, which requires the two peers (e.g., A and B) to have active sessions with the rendezvous server. This may lead to additional overhead and latency.

Furthermore, no single traversal technique works with all existing NATs because NAT behavior is not standardized [23]. An alternative to help peers behind NATs to be integrated in Cuckoo is deploying proxies. For posting, the Cuckoo peers behind NATs first send tweets to proxy servers and authorize the servers to deliver these messages via the social or media network in Cuckoo. On the other hand, the proxy servers wait/receive new tweets, and then send back to these peers behind NATs.

B. Support for SMS-Based Phone Clients

There are two kinds of solutions to support text message (e.g., SMS) based phone clients without Internet access. One potential solution is to let each user correspond to a paid compute utility running on the cloud (e.g., Amazon EC2, Rackspace Cloud Servers), which is proposed in [52]. Each personal "cloudlet" runs a Cuckoo peer sending and receiving tweets for the user. Background web services can be employed on the "cloudlets" to realize interactions with phone clients by short messages with special short codes like 10958 and 40404 in Twitter. These highly available compute utilities could not only solve the mobility usage but also tremendously improve the performance of Cuckoo. However, considering the free services, hosting a virtual machine (VM) in a cloud is currently costly (e.g., running a VM machine in Amazon EC2 costs upwards of US$75 per month). Another more practical solution is to employ proxy servers to achieve the support for phone clients, which is similar with the proxy-based solution for NAT support. For posting, phone clients first send tweets to proxy servers and let the server deliver these contents via the social or media network in Cuckoo. Meanwhile, the proxy servers are responsible for fetching information from cloud servers in response of mobile queries. Further, SMS providers can also be involved in Cuckoo architecture using the proxy-based solution, which can significantly improve the support for SMS and mobile services.

C. Support for Browser-Based Clients

Currently, Cuckoo peer is only implemented as independent client software. For browser-based users, the solution is to implement Cuckoo as web-browser plugins/add-ons which can be integrated in extendable web browsers like Firefox, Chrome, etc. Since Cuckoo is implemented using Java language which is very friendly to web programming, we believe that it is viable to shift/rewrite current Cuckoo code to plugin-based source code.

VII. Conclusion

In this paper, we have presented Cuckoo, a novel system architecture designed for scalable microblogging services. Based on the observation of microblogging's divergent traffic demands, Cuckoo decouples the dual functionality components of microblogging services, i.e., the online social network and the news delivery medium. We use complementary mechanisms for reliable content delivery while offloading processing and bandwidth costs away from a small centralized server base. We have prototyped Cuckoo and evaluated our prototype using trace-driven emulation over 30,000 Twitter users. Compared with the centralized architecture, Cuckoo achieves server bandwidth saving of 30-50%, CPU reduction of 50-60%, memory reduction of up to 50%, while guaranteeing reliable message delivery. In short, Cuckoo provides good performance for microblogging both as a social network and as a news media.

References

[1] TIB/Rendezvous, 1999. TIBCO white paper.

[2] AFP.com. 'Twitters' Beat Media in Reporting China Earthquake, 2008.

[3] Birman, K., Cooper, R., Joseph, T., Marzullo, K., Makpangou, M., Kane, K., Schmuck, F., and Wood, M. The Isis System Manual. Tech. rep., Cornell University, 1990.

[4] Bishop, M., Rao, S., and Sripanidkulchai, K. Considering Priority in Overlay Multicast Protocols under Heterogeneous Environments. In *Proc. of INFOCOM* (2006).

[5] Blake, C., and Rodrigues, R. High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two. In *Proc. of HotOS IX* (2003).

[6] Broadwell, P. M. Response Time as a Performability Metric for Online Services. Tech. Rep. UCB/CSD-04-1324, University of California at Berkeley, 2004.

[7] Buchegger, S., Schiöberg, D., Vu, L.-H., and Datta, A. PeerSoN: P2P Social Networking – Early Experiences and Insights. In *Proc. of SNS* (2009).

[8] Cabera, L. F., Jones, M. B., and Theimer, M. Herald: Achieving a Global Event Notification Service. In *Prof. of HotOS* (2001).

[9] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems 19*, 3 (2001), 332–383.

[10] Castro, M., Costa, M., and Rowstron, A. Debunking Some Myths about Structured and Unstructured Overlays. In *Proc. of NSDI* (2005).

[11] Castro, M., Jones, M. B., Kermarrec, A.-M., Rowstron, A., Theimer, M., Wang, H., and Wolman, A. An Evaluation of Scalable Application-level Multicast Built Using Peer-to-Peer Overlays. In *Proc. of INFOCOM* (2003).

[12] Cha, M., Haddadi, H., Benevenuto, F., and Gummadi, K. P. Measuring User Influence in Twitter: The Million Follower Fallacy. In *Proc. of ICWSM* (2010).

[13] Chan, C.-Y., Felber, P., Garofalakis, M., and Rastogi, R. Efficient Filtering of XML Documents with XPath Expressions. In *Proc. of ICDE* (2002).

[14] CNET News. Twitter Crippled by Denial-of-Service Attack, 2009.

[15] CUTILLO, L. A., MOLVA, R., AND STRUFE, T. Safebook: A Privacy Preserving Online Social Network Leveraging on Real-Life Trust. *IEEE Communication Magazine 47*, 12 (2009), 94–101.

[16] DABEK, F., ZHAO, B., DRUSCHEL, P., KUBIATOWICZ, J., AND STOICA, I. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proc. of IPTPS* (2003).

[17] DIAO, Y., RIZVI, S., AND FRANKLIN, M. Towards an Internet-Scale XML Dissemination Service. In *Proc. of VLDB* (2004).

[18] EUGSTER, P. Type-Based Publish/Subscribe: Concepts and Experiences. *ACM Transactions on Programming Languages and Systems (TOPLAS) 29*, 1 (2007).

[19] EUGSTER, P. T., FELBER, P. A., GUERRAOUI, R., AND KERMARREC, A.-M. The Many Faces of Publish/Subscribe. *ACM Computing Survey 35*, 2 (2003), 114–131.

[20] EUGSTER, P. T., GUERRAOUI, R., KERMARREC, A. M., AND MASSOULIÉ, L. From Epidemics to Distributed Computing. *IEEE Computer 37* (2004), 60–67.

[21] EXAMINER.COM. San Francisco Twitter Users Shocked to Lose All Their Followers, 2010.

[22] FASTCOMPANY.COM. Twitter Predicts Box-Office Sales Better Than a Prediction Market, 2010.

[23] FORD, B., SRISURESH, P., AND KEGEL, D. Peer-to-Peer Communication Across Network Address Translators. In *Prof. of USENIX ATC* (2005).

[24] FREY, D., GUERRAOUI, R., KERMARREC, A.-M., KOLDEHOFE, B., MOGENSEN, M., MONOD, M., AND QUÉMA, V. Heterogeneous Gossip. In *Proc. of Middleware* (2009).

[25] GHOSH, S., KORLAM, G., AND GANGULY, N. The Effects of Restrictions on Number of Connections in OSNs: A Case-Study on Twitter. In *Proc. of WOSN* (2010).

[26] GUARDIAN.COM. Twitter Election Predictions Are More Accurate Than YouGov, 2010.

[27] GYARMATI, L., AND TRINH, T. A. Measuring User Behavior in Online Social Networks. *IEEE Network Magazine, Special Issue on Online Social Networks 24*, 5 (2010), 26–31.

[28] JANSEN, B. J., ZHANG, M., SOBEL, K., AND CHOWDURY, A. Micro-blogging as Online Word of Mouth Branding. In *CHI Spotlight on Works in Progress* (2009).

[29] JAVA, A., SONG, X., FININ, T., AND TSENG, B. Why We Twitter: Understanding Microblogging Usage and Communities. In *Proc. of WEBKDD/SNA-KDD* (2007).

[30] JENKINS, K., HOPKINSON, K., AND BIRMAN, K. A Gossip Protocol for Subgroup Multicast. In *Proc. of International Workshop on Applied Reliable Group Communication* (2001).

[31] KERMARREC, A.-M., MASSOULIÉ, L., AND GANESH, A. J. Probabilistic Reliable Dissemination in Large-Scale Systems. *IEEE Transactions on Parallel and Distributed Systems 14*, 3 (2003), 248–258.

[32] KRISHNAMURTHY, B., GILL, P., AND ARLITT, M. A Few Chirps about Twitter. In *Proc. of WOSN* (2008).

[33] KRYCZKA, M., CUEVAS, R., GUERRERO, C., YONEKI, E., AND AZCORRA, A. A First Step Towards User Assisted Online Social Networks. In *Proc. of SNS* (2010).

[34] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is Twitter, a Social Network or a News Media? In *Proc. of WWW* (2010).

[35] LOO, B. T., HUEBSCH, R., STOICA, I., AND HELLERSTEIN, J. M. The Case for a Hybrid P2P Search Infrastructure. In *Proc. of IPTPS* (2004).

[36] MATHY, R., MATTHEWS, P., AND ROSENBERG, J. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN), 2010. RFC 5766.

[37] MISLOVE, A., MARCON, M., GUMMADI, K. P., DRUSCHEL, P., AND BHATTACHARJEE, B. Measurement and Analysis of Online Social Networks. In *Proc. of IMC* (2007).

[38] MISLOVE, A., POST, A., DRUSCHEL, P., AND GUMMADI, K. P. Ostra: Leveraging Trust to Thwart Unwanted Communication. In *Proc. of NSDI* (2008).

[39] MOBILESYRUP.COM. Twitter CEO: "40 Percent of All Tweets Created on Mobile Devices", 2011.

[40] MOTOYAMA, M., MEEDER, B., LEVCHENKO, K., VOELKER, G. M., AND SAVAGE, S. Measuring Online Service Availability Using Twitter. In *Proc. of WOSN* (2010).

[41] NEW YORK TIMES. Sports Fans Break Records on Twitter, 2010.

[42] RAMASUBRAMANIAN, V., PETERSON, R., AND SIRER, E. G. Corona: A High Performance Publish-Subscribe System for the World Wide Web. In *Proc. of NSDI* (2006).

[43] RATNASAMY, S., HANDLEY, M., KARP, R., AND SHENKER, S. Application-Level Multicast Using Content-Addressable Networks. In *Proc. of NGC* (2001).

[44] REUTERS NEWS. Twitter Snags over 100 Million Users, Eyes Money-Making, 2010.

[45] ROSENBERG, J. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols, 2010. RFC 5245.

[46] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. of Middleware* (2001).

[47] ROWSTRON, A., AND DRUSCHEL, P. Storage Management and Caching in PAST, A Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Proc. of SOSP* (2001).

[48] ROYAL PINGDOM. Twitter Growing Pains Cause Lots of Downtime in 2007, 2007.

[49] SANDLER, D., MISLOVE, A., POST, A., AND DRUSCHEL, P. FeedTree: Sharing Web Micronews with Peer-to-Peer Event Notification. In *Proc. of IPTPS* (2005).

[50] SANDLER, D. R., AND WALLACH, D. S. Birds of a FETHR: Open, Decentralized Micropublishing. In *Proc. of IPTPS* (2009).

[51] SHAKIMOV, A., LIM, H., CÁCERES, R., COX, L. P., LI, K., LIU, D., AND VARSHAVSKY, A. Vis-à-Vis: Privacy-Preserving Online Social Networking via Virtual Individual Servers. In *Proc. of COMSNETS* (2011).

[52] SHAKIMOV, A., VARSHAVSKY, A., COX, L. P., AND CÁCERES, R. Privacy, Cost, and Availability Tradeoffs in Decentralized OSNs. In *Proc. of WOSN* (2009).

[53] SRISURESH, P., FORD, B., AND KEGEL, D. State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs), 2008. RFC 5128.

[54] TWITTER BLOG. Big Goals, Big Game, Big Records, 2010.

[55] TWITTER BLOG. What's Happening with Twitter?, 2010.

[56] TWITTER ENGINEERING BLOG. A Perfect Storm.....of Whales, 2010.

[57] VAN RENESSE, R., BIRMAN, K. P., AND VOGELS, W. Astrolabe: A Robust and Scalable Technology For Distributed Systems Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems 21*, 3 (2003).

[58] WANG, F., XIONG, Y., AND LIU, J. mTreebone: A Collaborative Tree-Mesh Overlay Network for Multicast Video Streaming. *IEEE Transactions on Parallel and Distributed Systems 21*, 3 (2010), 379–392.

[59] XU, T., CHEN, Y., FU, X., AND HUI, P. Twittering by Cuckoo – Decentralized and Socio-Aware Online Microblogging Services. In *SIGCOMM Demo* (2010).

[60] XU, T., CHEN, Y., ZHAO, J., AND FU, X. Cuckoo: Towards Decentralized, Socio-Aware Online Microblogging Services and Data Measurement. In *Proc. of HotPlanet* (2010).

[61] ZAHARIA, M., AND KESHAV, S. Gossip-based Search Selection in Hybrid Peer-to-Peer Networks. In *Proc. of IPTPS* (2005).