

Scaling Microblogging Services with Divergent Traffic Demands

Tianyin Xu¹, Yang Chen¹, Lei Jiao¹, Ben Y. Zhao²,
Pan Hui³, and Xiaoming Fu¹

¹ University of Goettingen

² U.C. Santa Barbara

³ Deutsche Telekom Laboratories

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.

1 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, the most successful service, has more than 100 million users and generates more than 65 million “tweets” per day [23, 29]. 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 [21].

While originally designed as an online social network (OSN) for users with quick updates, the 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 [9], even

producing effective predictions for election results [12]. Still other projects have demonstrated that Twitter traffic can be mined as a sensor for Internet service failures [20] and even a real-time warning system for earthquakes [1].

In addition to these applications, Twitter usage by its users 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 personalities to their fans and followers [4, 17]. These asymmetric communication channels more closely resemble news media outlets than social communication channels. Because of the popularity of celebrities on Twitter (e.g., Lady Gaga, Britney Spears, and Justin Bieber account for over 30 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 the efforts to scale the system, Twitter has suffered significant loss in availability from malicious attacks and hardware failures [5, 8], and more frequently from traffic overload and flash crowds [25, 30, 31]. As short-term solutions, Twitter has employed per-user request and connection limits [11], as well as network usage monitoring and doubling the capacity of internal networks [31], 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, 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 [19].

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.

¹ We first outlined our idea in an earlier workshop paper [35].

We have implemented a Cuckoo prototype and made its source code and datasets publicly available². We evaluated a small-scale deployment for 50 Twitter users running on 5 laptops as a demonstration [33]. In addition, we have conducted laboratory experiments using a detailed Twitter trace containing 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 between Cuckoo peers.

In summary, this paper makes three key contributions:

1. A novel system architecture 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 savings, CPU and memory reduction, as well as reliable message delivery and efficient micronews dissemination.

2 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.

Microblogging Model. The common model of microblogging services is the simplified publish-subscribe (Pub-Sub) model (c.f., [34]) based on the “follow” operation. The microblogging model is deceptively simple: The 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 all his (or her) tweets, i.e., being a *follower* means that the user will receive all the news from the *followees*. Currently, the microblogging model is implemented by using naïve polling for detecting updates in the centralized architecture.

There are several prior works on Pub-Sub systems that abandon the use of naïve polling, thus achieving high scalability and performance [22, 26]. Their key idea is cooperative polling between dedicated middleware mediators, named *brokers*. Microblogging differentiates from the traditional Pub-Sub systems by the system architecture. In microblogging, there is no always-on broker that collects events from publishers and sends notifications to subscribers. The key problem of microblogging is how to directly deliver publishers’ tweets to their followers with divergent traffic demands. Cuckoo shares the insight with the prior works that the blind polling is the prime culprit of poor performance and limited scalability. Instead, Cuckoo enables user clients to share tweets in the peer-assisted fashion. On the other hand, Cuckoo interoperates with the current polling-based web architecture, requiring no change to legacy web servers.

Microblogging Measurement and Analysis. Microblogging services are widely recognized as online social network services for the explicit and implicit

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

social relations [11, 14, 16, 20]. For example, users exhibiting reciprocity (i.e., following each other) should be acquaintances, typical in OSNs. According to the “follow” relations, Krishnamurthy et al. identify distinct groups of users, e.g., broadcasters and evangelists [16]. 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 [11]. Java et al. report early observations of Twitter and analyze social communities formed by users with similar interests [14]. On the other hand, some researchers recently argue that microblogging, as exemplified by Twitter, serves more as news media outlets than OSN services [4, 17, 27]. Due to the one-sided nature of the “follow” relation, there are a small number of highly-subscribed users (e.g., celebrities and mass media) who have large numbers of followers and post far more tweets than the other users. These users generate the greatest per-capita proportion of network traffic and trend the trends.

One of Cuckoo’s design rationales is to separate microblogging’s dual components, 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.

Decentralized Microblogging and OSN Systems. There are several decentralized OSN systems proposed for different research concerns. FETHR [27] 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 component nor implement it in its prototype. Other practical issues such as client heterogeneity support are also missing in FETHR. PeerSoN [2] is a prototype of P2P OSNs that uses encryption to protect user privacy against OSN providers and third-party applications. It uses dedicated DHT for data lookup, based on which direct user information exchanging can be achieved. Vis-à-Vis [28] is based on the concept of VIS, 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 representing OSN groups, with one DHT for each group. Safebook [6] is a decentralized OSN that aims at protecting users’ security and privacy based on trust transitivity.

Cuckoo proposes a new system architecture tailored for microblogging services. It consists of two overlay networks with different content delivery mechanisms. 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.

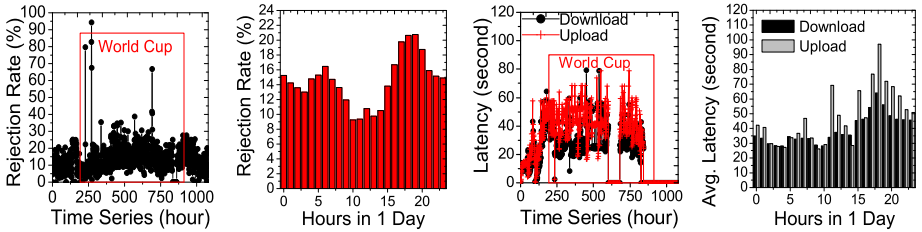


Fig. 1. Twitter measurement: Service rejection (a) in time series (Jun. 4–18, 2010), (b) in 24 hours; Response latency (c) in time series (Jun. 4–18, 2010), (d) in 24 hours

3 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 [34] measurement and a user behavior analysis for over 300,000 Twitter users. In addition, we measure the system scalability of the generic centralized microblogging architecture.

Availability and Performability of Twitter. We first conducted a measurement study on the availability and performability of Twitter in terms of service rejection 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 regarded as Twitter’s worst month since October 2009 from a site stability and service outage perspective [30]. We used JTwitter as the Twitter API to do the measurement.

For service rejection 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 rejection event. Fig. 1(a) shows the average service rejection rate per hour during our measurement period. We see that Twitter’s availability was poor – the rejection rate was already about 10% in normal time. Moreover, the flash crowd caused by FIFA World Cup made an obvious impact on service rejection 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(b) reports the average rejection 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 rejection.

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

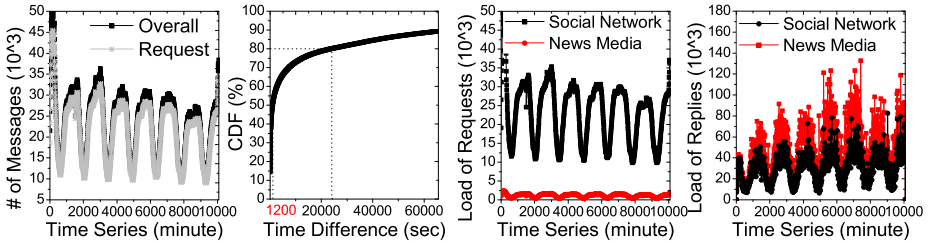


Fig. 2. User access patterns: (a) # of request messages to the servers; (b) Time differences of two adjacent tweets; (c) Incoming traffic load; (d) Outgoing traffic load

latency. Such round was repeated every 5 seconds. Similar as Fig. 1(a) and 1(b), Fig. 1(c) and 1(d) shows the measured response latency of the Twitter service³. No surprisingly, Twitter’s performability, in terms of response latency, was unsatisfactory 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 [30, 31], their solutions include doubling the capacity, monitoring, and rebalancing the traffic on their internal networks, which do not scale well with the unprecedented growth.

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 [17] 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 1-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*. The polling period is set as one minute according to the setting options of common Twitter clients (e.g., Ambientweet, Gwibber, Osfoora) [34]. For session durations, we use the duration dataset provided in [13]. The details of the above datasets and data processing are described in Section 5.1.

Fig. 2(a) shows the 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. 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. 2(b) 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 the polling requests are fully scheduled in these intervals, the resource waste due to unnecessary traffic is tremendous.

³ The gaps in Fig. 1(c) is due to server cutoffs during the measurement period.

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 [4, 17, 27] 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 [17] 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. 2(c) shows the incoming traffic load in terms of received messages. For a 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. 2(c), we find that the social network usage occupies the dominant proportion of incoming traffic load – about 95% of incoming load is for social network usage while less than 5% is for news media. Fig. 2(d) reports the outgoing traffic 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 social user. We can see from Fig. 2(d) that although news media usage holds small proportion of server requests (Fig. 2(c)), it occupies a great proportion of outgoing traffic load, with 1.66 times on average more than the proportion of social network usage. Thus, the dual functionality components of microblogging have divergent traffic patterns, and the mix of them at the same time makes the system using a single dissemination mechanism hard to scale.

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 tweet message, and then receives ACK to display. On the other hand, users detect updates by periodically polling through established connections.

We use the Twitter trace described in the previous section 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.

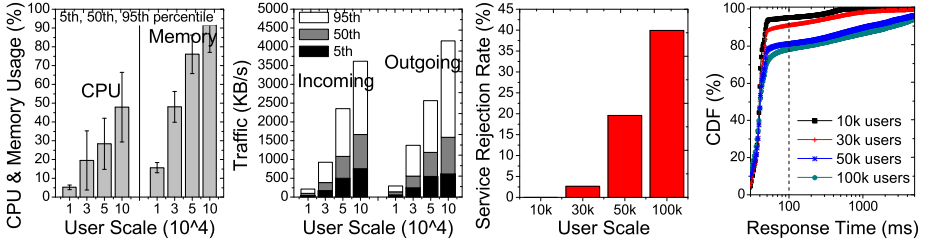


Fig. 3. Scalability of the centralized microblogging architecture: (a) CPU and memory usage; (b) Traffic usage; (c) Service rejection rate; (d) Response latency

Fig. 3 demonstrates the limited system scalability of the centralized architecture in terms of CPU and memory usage, traffic usage, service rejection rate, and response time. For CPU usage, memory usage, and traffic usage (Fig. 3(a) and Fig. 3(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. 3(c) shows service rejection 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 rejection 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 rejection rate is lower than 5%. But for 100,000 users, even in regular time, the servers are overloaded so that the rejection rate is extremely high. Fig. 3(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% of requests are satisfied within 100 ms. While the user scale increases to 30,000, only 80% of 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 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 the 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 burden and traffic waste towards high scalability by abandoning polling and decoupling the dual functionality components.

4 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.

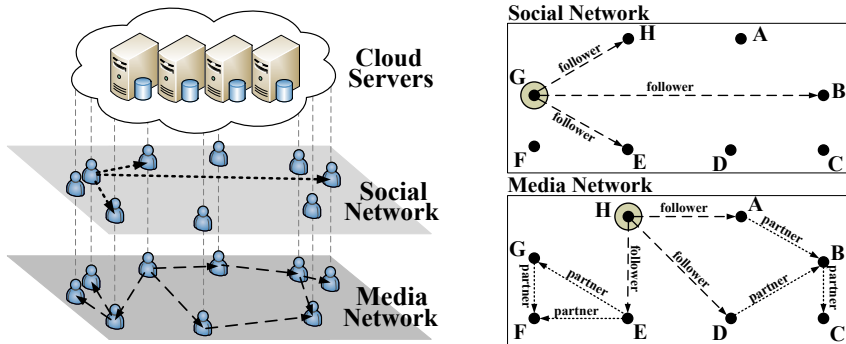


Fig. 4. Cuckoo: (a) System architecture; (b) Complementary content delivery

4.1 System Architecture

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 divergent traffic and workload patterns due to their different dissemination models. As discussed in Section 3, although the social network component occupies more than 95% request load, the news media component holds greater proportion 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 [27] 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 [27] 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 Twitter-sphere [17] 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 or mass media. Besides, media users post tweets (named *micronews*) much more frequently than social users. The correlation analysis in [17] shows that the number of tweets grows by an order of magnitude for the users with 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. 4(a) shows the high-level architecture of Cuckoo which 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 the unicast fashion. For the news media component, Cuckoo peers with the same interest 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 numbers 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 numbers 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. 4(b) demonstrates Cuckoo’s complementary content delivery mechanisms corresponding to the two overlay networks. Besides, Cuckoo employs a set of stable peers to form a DHT (Distributed Hash Table), e.g., [24, 36] that maintains all the users’ connection information (e.g., IP address, port) named *node handlers* (NHs). 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.

Combination of Server Cloud and Client Peers. As shown in Section 3, 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 traditional centralized client-server architecture is hard to scale. Meanwhile, truly decentralized P2P systems have earned notoriety for the difficulties coping with availability and consistency, and thus achieved limited success in the past. For example, FETHR [27] provides no guarantee on data delivery. The 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, the Cuckoo peers also upload their 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 period. By abandoning naïve polling and offloading the dissemination operation cost, the server cloud gets rid of the main server load towards high 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 [25], under attack [5]), Cuckoo peers can still find and communicate with each other. Moreover, information loss in a single location [8] 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 polling-based web architecture, requiring no special feature on the server side.

4.2 Social Relations

We describe how Cuckoo peers maintain the social relations to form the two overlay networks, and how “follow” is operated to build these relations.

Social Relation Maintenance. In typical microblogging services, a user has the following social relations: *followee*, *follower*, and *partner*. To form the social network, each peer maintains the followee and follower information in its local database. The follower information 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 e_i in user i ’s follower list can be presented as: $e_i = \max(\min(F_i, H), \log(F_i))$, where F_i denotes the number of followers of user i , and H is the threshold to separate social users and media users. To form the media network, the Cuckoo peer maintains sets of partners corresponding to the media users 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 groups, which to some extent suppresses the behavior of evangelists (e.g., spammers or stalkers) [16].

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, the Cuckoo peer first lookups the followee’s NH according to his (or her) `userId`⁴ 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 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. 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.

⁴ In Cuckoo, each user is assigned a unique `userId` by the server cloud at registration, which simplifies the authentication and Id assignment.

4.3 Unicast Delivery for the Social Network

When a user posts a new tweet, the microblogging service should guarantee that all the users' 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 can search up-to-date NHs.

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 being 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 missing parts can be efficiently detected by giving his (or her) last departure time or the `statusId`⁵ 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 maintained in the DHT.

4.4 Gossip Dissemination for the Media Network

In the media network, media users cannot afford sending updates to all their followers. In this case, Cuckoo uses gossip-based dissemination, i.e., enable interested users to be involved in the micronews dissemination process. Gossip information dissemination has been proved to be scalable and resilient to network dynamics. The theoretical support provided in [15] 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.

To discovery online partners in case of churn, we design a DHT-based partner collection mechanism which is elaborated in [34]. We sketch our basic idea as follows. The 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., its media followees' `userIds`). This hello message is routed to the DHT node with `nodeId` numerically closest to X . Meanwhile, 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

⁵ In Cuckoo, each tweet is bounded to a unique `statusId`.

of media users. To look for new online partners of a media user, a Cuckoo peer uses DHT-based random walks [3] to collect partners over the DHT topology by checking these indices. The DHT-based random walk ensures that nodes in the DHT are visited only once during a collection process. Since the media users have high popularity, the random walk-based collection is efficient [34].

Cuckoo adopts the simple “infect and die” model [7] 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` of the tweet. For a new tweet, 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 micronews is disseminated within the circle of interested peers, i.e., the followers. The number of gossip rounds R , i.e., the network hops necessary to spread a micronews to all the online followers respects the equation [7]: $R = \frac{\log(n)}{\log(\log(n))} + O(1)$, which shows that it takes at most a logarithmic number of steps for a micronews to reach every online follower.

4.5 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 [19]. Thus, it is important to support client heterogeneity in Cuckoo, considering the economic burden of increased load on mobile peers such as higher cost for network usage (due to expensive or limited mobile data plans), and 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 can be integrated to identify stable nodes in overlay networks, e.g., the nodes already with higher ages tend to stay longer [32]. 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 and the main difference between them is that Cuckoo-Mobile peers do not participate in the gossip dissemination process in the media network while the Cuckoo-Lite do (as the Cuckoo-Comp). 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*. We call both Cuckoo-Comp and Cuckoo-Lite peers as *gossip nodes* that can use heterogeneity-aware gossip (e.g., [10]) to tune the fanout. The dissemination is initiated from the publisher, gossiped through the Cuckoo-Comp and Cuckoo-Lite peers, and simultaneously spread to Cuckoo-Mobile peers. The details of dissemination with heterogeneous clients can be found in [34] including the solutions to support text message based phone clients and clients behind NATs (Network Address Translations).

4.6 Message Loss Detection and Security Issues

Detecting Lost Tweets. While Cuckoo’s gossip-based probabilistic dissemination achieves high resilience to node failures as well as high coverage rate, it provides no guarantee that each tweet is reached to all the followers 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, the 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 [34] or the server cloud.

Security. The presence of malicious users (e.g., attackers, spammers) requires additional components to safeguard against various kinds of security threats. Although the main concern of this paper is not on the security aspect, we introduce the basic ideas of Cuckoo’s security components in this section and refer the details to [34]. To defend against spam distributed by the malware that impersonate normal users, the digital signature based on asymmetric key cryptography is attached within each message for authentication. Moreover, these digital signatures are capable of defending against violating message integrity by altering the contents of tweets. For DoS attacks and content censorships that target on crippling the server cloud [5], Cuckoo exploits its distributed nature: peers can still deliver/disseminate messages via the social/media network, and mark the blocked tweets for further uploading. For the brute-force attacks where malicious nodes generate unwanted traffic to harass normal peers’ operations, trust and reputation mechanisms can be employed. The social relations maintained on each peer provide nature trust relationships to build the reputation model, based on which unwanted communications can be thwarted [18].

5 Experimental Evaluation

To evaluate Cuckoo’s performance, we run Cuckoo prototype using trace-driven emulation of the Twitter trace containing 30,000 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, as well as the performance of message sharing and micronews dissemination between Cuckoo peers.

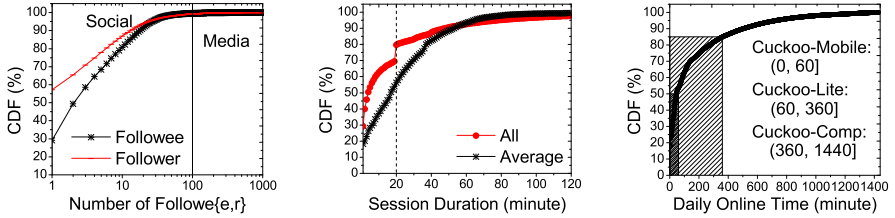


Fig. 5. Dataset: (a) Number of followee{e, r}s; (b) Session duration; (c) Online time

5.1 Experiment Settings

Dataset. We use the raw dataset described in Section 3 to evaluate Cuckoo’s performance. The raw dataset contains 3,117,750 users’ profiles, social links, and all the tweets maintained on the Twitter site⁶ from Mar. 6 to Apr. 2, 2010. Still, we focus on user access patterns during the 1-week period from Feb. 1 to Feb. 7, 2010. We use BFS with the start user Ustream to create the experiment dataset containing 30,000 users’ information to match the capacity of the emulation testbed. For social links, we prune the irrelevant links outside the dataset. Fig. 5(a) shows the CDF of users’ followee/follower number. We separate the social users and media users according to each user’s follower number with the threshold H as 100 instead of 1000 in Section 3 due to the social link pruning, and get 29,874 (99.58%) social users and 126 (0.42%) media users. We use the OSN session dataset provided by [13] to conservatively emulate Cuckoo users’ session durations because so far no microblogging service provides user session information or user online/offline status information. Fig. 5(b) plots the CDF of the average session duration of each user and the CDF of all the durations. We classify the three types of Cuckoo users according to their daily online time, i.e., Cuckoo-Comp users are those whose daily online time exceeds 360 minutes, and Cuckoo-Mobile users spend less than 60 minutes online per day, and the others are Cuckoo-Lite users. Fig. 5(c) shows the CDF of the daily online time and the classification. We can see that about 50% Cuckoo peers are Cuckoo-Mobile clients. The details of data processing and dataset analysis can be found in [34].

Implementation. We have built a Cuckoo prototype 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 with different software components. We use Java socket to implement end-to-end message delivery, and define different types of application-layer messages. For local data management, we use XML to store node states and social links including followee/follower profiles, followee tweets, and self-posted tweets. We choose Pastry [24] and its 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 special feature (e.g., leaf set), so it is applicable for any structured

⁶ Twitter only reserves about 3000 tweets per user and discards the previous tweets.

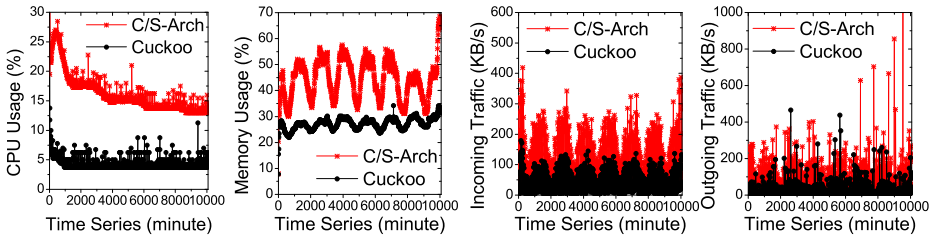


Fig. 6. Resource usage of the server cloud in time series (from Feb. 1 to Feb. 7, 2010): (a) CPU; (b) Memory; (c) Incoming traffic; (d) Outgoing traffic

overlay that supports the key-based routing. The server cloud prototype adds up to 1500 lines of Java code. It uses plain text files to store all users’ information.

Deployment. We deploy 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 RAM, and 9 Dell Optiplex 755, each with two 3.00 Ghz Intel Core 2 Duo CPU and 3.6 GB RAM. We deploy four servers to build the server cloud on another Dell PowerEdge T300 machine and let them share storage, so that these servers have no inconsistency problem. We locate these machines into two LANs connected by a 10 Gb/s Ethernet cable. We run the Cuckoo peers based on the 1-week Twitter trace described in the previous section. 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.

5.2 Server Cloud Performance

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 (c.f., Section 3), 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.

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. 6(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 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. 6(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. As a result of message overhead savings, Cuckoo effectively reduces the memory

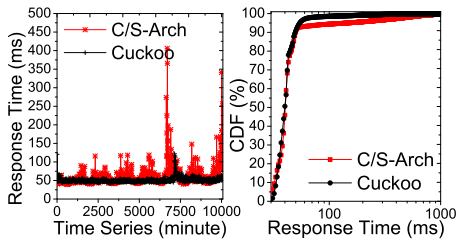


Fig. 7. Response latency (a) in time series, (b) in CDF

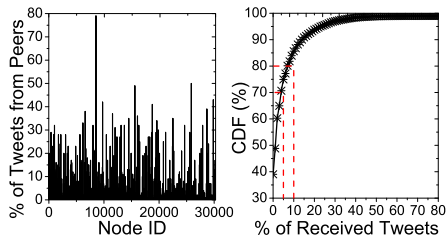


Fig. 8. Percentage of received tweets (a) in time series, (b) in CDF

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% of memory usage reduction for the server cloud at peak/leisure time. Fig. 6(c) and Fig. 6(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 for C/S-Arch is 300 KB/s, 400 KB/s at peak time and 400 KB/s, 100 KB/s at leisure time. Thus, the server cloud of Cuckoo saves about 50% of bandwidth consumed for both incoming and outgoing traffic compared with that of C/S-Arch.

Response Latency. We examine the response latency provided by the server cloud of Cuckoo and C/S-Arch. Fig. 7(a) shows the response latency in time series and Fig. 7(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 of message overhead is smaller than that in 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 server cloud of C/S-Arch has too much burden so that it can hardly satisfy all the concurrent requests at the same time.

5.3 Cuckoo Peer Performance

In this section, we characterize the performance of Cuckoo peers. Each peer maintains a message log that records all the received, sent, and forwarded messages. By collecting these logs from Cuckoo peers, we analyze the performance of message sharing. Moreover, the message logs provide the detailed information of

disseminated messages including `statusIds` of tweets, network hops, and redundancy. Based on these, we analyze the performance of micronews dissemination.

Message Sharing. Fig. 8 shows the performance of message sharing between Cuckoo peers. Fig. 8(a) shows for each peer, the percentage of tweets received from other Cuckoo peers other than from the server cloud, while Fig. 8(b) is the CDF of the percentages. According to Fig. 8(a) and 8(b), around 30% users get more than 5% of their overall subscribed tweets from other peers, and around 20% get more than 10%. 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. 5(b)) [13], 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 the better performance of message sharing in Cuckoo.

Micronews 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. 5(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 [34] 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. 5(c)), 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. 9 demonstrates the performance and overhead of Cuckoo's micronews dissemination in terms of coverage, network 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 rate. Fig. 9(a) shows the coverage rate of each dissemination process in time series. In this figure, the media user Ustream, Etsy, jalenrose with 29,927, 6,914, 696 followers publishes

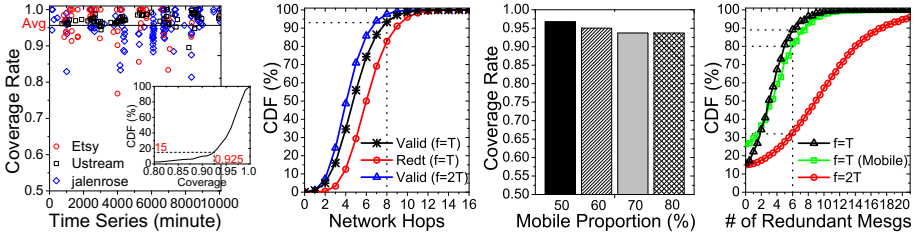


Fig. 9. Micronews dissemination: (a) Coverage; (b) Average network hops; (c) Coverage rate with different proportion of mobile nodes; (d) Average redundant messages

61, 59, 117 tweets respectively in the one-week period. We can see from the CDF that around 85% dissemination processes cover more than 92.5% of 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 is only few process with coverage rate lower than 80% due to the uncertainty of gossip and user asynchronism. Fig. 9(b) shows the CDF of the network hops of tweets received by all the users. We compare the network hops of *valid* tweets (i.e., tweets received for first time) with those of *redundant* (marked as “Redt” in Fig. 9(b)) 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. 9(c) 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 rate 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 of 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 [34].

Client Overhead. In Cuckoo, the client overhead for message delivery 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. 9(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 more than 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).

6 Conclusion

We have presented Cuckoo, a novel system architecture designed for scalable microblogging services. Based on the observation of divergent traffic demands, Cuckoo decouples the dual components of microblogging services. 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 notable server bandwidth savings, CPU and memory reduction, while guaranteeing reliable message delivery. In short, Cuckoo provides good performance for microblogging both as a social network and as a news media.

Acknowledgements. We thank our shepherd Ranveer Chandra and the anonymous reviewers for their valuable feedback. This work is supported in part by the National Science Foundation under grants IIS-0916307 and IIS-847925.

References

1. AFP.com. ‘Twitters’ Beat Media in Reporting China Earthquake (2008)
2. Buchegger, S., Schiöberg, D., Vu, L.-H., Datta, A.: PeerSoN: P2P Social Networking – Early Experiences and Insights. In: Proc. of SNS (2009)
3. Castro, M., Costa, M., Rowstron, A.: Debunking Some Myths about Structured and Unstructured Overlays. In: Proc. of NSDI (2005)
4. Cha, M., Haddadi, H., Benevenuto, F., Gummadi, K.P.: Measuring User Influence in Twitter: The Million Follower Fallacy. In: Proc. of ICWSM (2010)
5. CNET News. Twitter Crippled by Denial-of-Service Attack (2009)
6. Cutillo, L.A., Molva, R., Strufe, T.: Safebook: A Privacy Preserving Online Social Network Leveraging on Real-Life Trust. IEEE Communication Magazine 47(12), 94–101 (2009)
7. Eugster, P.T., Guerraoui, R., Kermarrec, A.M., Massoulié, L.: From Epidemics to Distributed Computing. IEEE Computer 37, 60–67 (2004)
8. Examiner.com. San Francisco Twitter Users Shocked to Lose All Their Followers (2010)
9. FastCompany.com. Twitter Predicts Box-Office Sales Better Than a Prediction Market (2010)
10. Frey, D., Guerraoui, R., Kermarrec, A.-M., Koldehove, B., Mogensen, M., Monod, M., Quéma, V.: Heterogeneous Gossip. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 42–61. Springer, Heidelberg (2009)
11. Ghosh, S., Korlam, G., Ganguly, N.: The Effects of Restrictions on Number of Connections in OSNs: A Case-Study on Twitter. In: Proc. of WOSN (2010)

12. Guardian.com. Twitter Election Predictions Are More Accurate Than YouGov (2010)
13. Gyarmati, L., Trinh, T.A.: Measuring User Behavior in Online Social Networks. *IEEE Network Magazine, Special Issue on Online Social Networks* 24(5), 26–31 (2010)
14. Java, A., Song, X., Finin, T., Tseng, B.: Why We Twitter: Understanding Microblogging Usage and Communities. In: *Proc. of WEBKDD/SNA-KDD* (2007)
15. Kermarrec, A.-M., Massoulié, L., Ganesh, A.: Probabilistic Reliable Dissemination in Large-Scale Systems. *IEEE Transactions on Parallel and Distributed Systems* 14(3), 248–258 (2003)
16. Krishnamurthy, B., Gill, P., Arlitt, M.: A Few Chirps about Twitter. In: *Proc. of WOSN* (2008)
17. Kwak, H., Lee, C., Park, H., Moon, S.: What is Twitter, a Social Network or a News Media? In: *Proc. of WWW* (2010)
18. Mislove, A., Post, A., Druschel, P., Gummadi, K.P.: Ostra: Leveraging Trust to Thwart Unwanted Communication. In: *Proc. of NSDI* (2008)
19. Mobilesyryp.com. Twitter CEO: 40 Percent of All Tweets Created on Mobile Devices (2011)
20. Motoyama, M., Meeder, B., Levchenko, K., Voelker, G.M., Savage, S.: Measuring Online Service Availability Using Twitter. In: *Proc. of WOSN* (2010)
21. New York Times. Sports Fans Break Records on Twitter (2010)
22. Ramasubramanian, V., Peterson, R., Sirer, E.G.: Corona: A High Performance Publish-Subscribe System for the World Wide Web. In: *Proc. of NSDI* (2006)
23. Reuters News. Twitter Snags over 100 Million Users, Eyes Money-Making (2010)
24. Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In: Liu, H. (ed.) *Middleware 2001*. LNCS, vol. 2218, pp. 329–351. Springer, Heidelberg (2001)
25. Pingdom, R.: Twitter Growing Pains Cause Lots of Downtime in 2007 (2007)
26. Sandler, D., Mislove, A., Post, A., Druschel, P.: FeedTree: Sharing Web Micronews with Peer-to-Peer Event Notification. In: van Renesse, R. (ed.) *IPTPS 2005*. LNCS, vol. 3640, pp. 141–151. Springer, Heidelberg (2005)
27. Sandler, D.R., Wallach, D.S.: Birds of a FETHR: Open, Decentralized Micropublishing. In: *Proc. of IPTPS* (2009)
28. Shakimov, A., Varshavsky, A., Cox, L.P., Cáceres, R.: Privacy, Cost, and Availability Tradeoffs in Decentralized OSNs. In: *Proc. of WOSN* (2009)
29. Twitter Blog. Big Goals, Big Game, Big Records (2010)
30. Twitter Blog. What’s Happening with Twitter? (2010)
31. Twitter Engineering Blog. A Perfect Storm....of Whales (2010)
32. Wang, F., Xiong, Y., Liu, J.: mTreebone: A Collaborative Tree-Mesh Overlay Network for Multicast Video Streaming. *IEEE Transactions on Parallel and Distributed Systems* 21(3), 379–392 (2010)
33. Xu, T., Chen, Y., Fu, X., Hui, P.: Twittering by Cuckoo – Decentralized and Socio-Aware Online Microblogging Services. In: *SIGCOMM Demo* (2010)
34. Xu, T., Chen, Y., Jiao, L., Zhao, B.Y., Hui, P., Fu, X.: Cuckoo: Scaling Microblogging Services with Divergent Traffic Demands. Technical Report IFI-TB-2011-01, Univ. of Goettingen (2011)
35. Xu, T., Chen, Y., Zhao, J., Fu, X.: Cuckoo: Towards Decentralized, Socio-Aware Online Microblogging Services and Data Measurement. In: *Proc. of HotPlanet* (2010)
36. Zhao, B.Y., Huang, L., Stribling, J., Rhea, S.C., Joseph, A.D., Kubiatowicz, J.D.: Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications* 22(1), 41–53 (2004)