Twitter in Disaster Mode:
Security Architecture

Theus Hossmann, Paolo Carta, Dominik
Schatzmann, Franck Legendre
Communication Systems Group
ETH Zurich, Switzerland
lastname@tik.ee.ethz.ch

Per Gunningberg, Christian Rohner
Communication Research Group
Uppsala University, Sweden
firstname.lastname@it.uu.se

ABSTRACT

Recent natural disasters (earthquakes, floods, etc.) have shown that people heavily use platforms like Twitter to communicate and organize in emergencies. However, the fixed infrastructure supporting such communications may be temporarily wiped out. In such situations, the phones' capabilities of infrastructure-less communication can fill in: By propagating data opportunistically (from phone to phone), tweets can still be spread, yet at the cost of delays.

In this paper, we present *Twimight* and its network security extensions. Twimight is an open source Twitter client for Android phones featured with a "disaster mode", which users enable upon losing connectivity. In the disaster mode, tweets are not sent to the Twitter server but stored on the phone, carried around as people move, and forwarded via Bluetooth when in proximity with other phones. However, switching from an online centralized application to a distributed and delay-tolerant service relying on opportunistic communication requires rethinking the security architecture. We propose security extensions to offer comparable security in the disaster mode as in the normal mode to protect Twimight from basic attacks. We also propose a simple, yet efficient, anti-spam scheme to avoid users from being flooded with spam. Finally, we present a preliminary empirical performance evaluation of Twimight.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Wireless Communications

General Terms

Design, Security

1. INTRODUCTION

In recent disaster events (e.g., earthquakes, floods), online social networks (OSNs) such as Facebook and Twitter have proven to be the communication tools of choice for affected people. Not only were they heavily used for spreading news and updates across the world [1] but also for connecting and organizing victims [2] locally at the disaster site. However, these platforms were not explicitly designed for emergency situations in the first place. In fact, their value in such environments could be greatly improved by extending them with features specially designed for hostile circumstances.

One drawback of today's OSNs in disaster situations is their dependance on fixed network infrastructure. If the wired and/or cellular infrastructure is wiped out by natural forces (or even just congested in the aftermath of a disaster) the client-server communication of current OSNs is not feasible. Thus, in the time until emergency response forces are able to repair infrastructure or deploy temporary communication solutions (e.g., wireless mesh networks, satellite phones) people are left without any means to communicate.

To overcome this limitation, the use of delay tolerant opportunistic networks [3, 4] has been proposed. The approach we use in this paper is based on these ideas. Our goal is to augment OSN applications – which the users already have installed on their mobile phones and use in their every day life – with an additional feature, the *disaster mode*. In particular, we implement the disaster mode in *Twimight*[1], our open source Twitter client. We choose Twitter since it has proven to be a highly useful communication platform in past emergency situations [1, 2]. Upon losing Internet connectivity, the user can simply enable the disaster mode to start *opportunistic communications* [5, 6]. Tweets entered in disaster mode then are stored locally on the phone, carried around as people move, and forwarded using Bluetooth as they are in proximity with other phones. Once connectivity is regained, the user's disaster tweets are uploaded to the Twitter server.

Such mobility-assisted opportunistic communication has the advantage of being ready instantly after losing connectivity. No deployment of temporary infrastructure is required, since people have everything that is need on their phones.

---

[1]Download: http://code.google.com/p/twimight

However, it has the disadvantage of being a purely best-effort technology and comes with potential delays in tweet delivery – depending on the density of people and on their mobility. Hence, the disaster mode is mainly suited to provide a communication means to people in a *first phase* (during the hours or few days after a disaster happens), *before* emergency response gets active and reaches the disaster site. In a *second phase*, once rescue missions have started and specialized forces are in place, there may be better suited technologies (e.g., satellite communication) to coordinate operations. We illustrate these two phases and the respective communication needs in Figure 1.

In this paper, we focus particularly on the security aspect of Twimight. Moving away from an online centralized service to a distributed and delay-tolerant service relying on opportunistic communications requires rethinking the security architecture. Since tweets are no longer exchanged directly with the Twitter servers but relayed potentially over multiple opportunistic hops, authenticity, integrity and confidentiality have to be guaranteed without accessing the server.

The contributions of this paper are the following:

- We describe the design and prototype of *Twimight*, our disaster Twitter client. It behaves like a normal Twitter client but includes a "disaster mode", enabling opportunistic communication where tweets spread epidemically (Section 2).

- We propose a security architecture to secure the dissemination of tweets in disaster mode. We introduce the Twimight Disaster Server (TDS) for managing and distributing cryptographic keys and certificates *before* a disaster happens, such that all components are in place once connectivity is lost. We also propose a simple, yet efficient strategy to throttle the dissemination of spam (Section 3).

- Finally, we validate the dependability of Twimight in disaster mode using small-scale experiments. We evaluate the performances of the opportunistic dissemination of tweets in terms of delay, hop count and impact on the battery consumption (Section 4).

As a final note, we want to highlight that one of the main design goal of Twimight is *simplicity*. We believe that the environment and circumstances of a natural disaster differ from case to case and it is hard to predict what exact features are needed in a future situation. Hence, we do not aim at providing a highly specialized and sophisticated solution. Instead, we want to let people use the tools they know from their everyday life, modifying them as little as possible. We trust that affected people will self-organize and figure out the right usage of the tool for a particular situation.

2. TWIMIGHT ARCHITECTURE

In this section we provide a brief overview of the architecture of Twimight with the main focus on the disaster mode
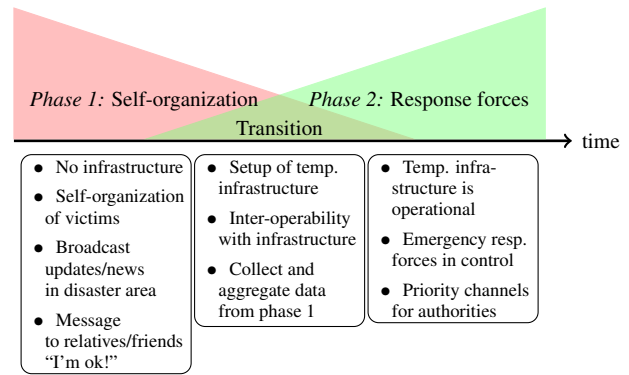

Figure 1: Two phases of disaster response.

and the opportunistic spreading of tweets[2]. We have implemented *Twimight* as a disaster ready Twitter client in Java for the Android operating system. Figure 3 gives an overview of the two modes of operations, normal and disaster, which we detail next. *Twimight* accesses user data using OAuth and the Twitter API after the user grants access to *Twimight* by logging in with its Twitter credentials.

Twimight in Normal mode: *Twimight* supports most basic Twitter functionality like showing a timeline (i.e., the most recent tweets of the followed users), sending tweets, re-tweeting and replying to and setting favorites, direct messages etc. In normal operation, *Twimight* queries Twitter for new tweets every 2 Minutes. The tweets obtained from the queries are cached locally in a SQLite database. Figure 2a shows the timeline in normal operation and the actions a user can take.
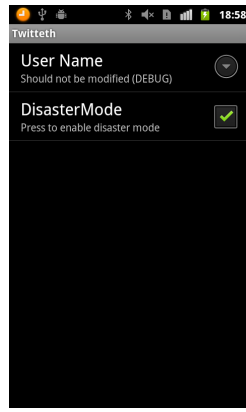
Twimight in Disaster mode: *Twimight* has a checkbox in the settings menu to enable the disaster mode as shown in Figure 2b. The normal Twitter client functionalities stay the same but now rely on opportunistic communications. Upon enabling the disaster mode, the tweets of the user are internally assigned the special status of *disaster tweets*. They are stored locally in a separate table for opportunistic spreading and later publication to the Twitter servers whenever connectivity is detected. Compared to the normal mode of operation of Twitter, disaster tweets are sent epidemically and are thus public to everyone. Disaster tweets are highlighted in red on the user interface (Figure 2c) to mark them as important regardless of the receiver being one of the sender's followers or not. The goal is that all tweets sent during a disaster can be received and displayed to as many people as possible.

In disaster mode, the device enables Bluetooth and maintains the CPU awake acquiring a wake lock to constantly listen to connection requests from other devices. Further, it scans periodically for reachable Bluetooth devices. The scanning interval has been set to 2 minutes $\pm U[0, 20]$ seconds. It represents a tradeoff between energy consumption and delay as a longer scanning interval would miss short
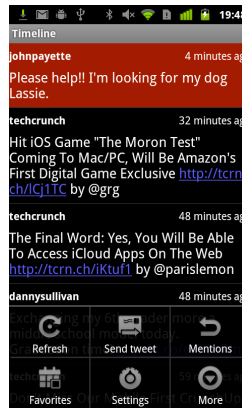

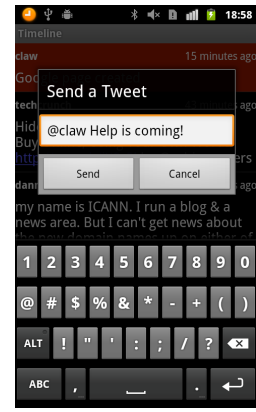[2]For a more detailed discussion of Twimight functionality see also [7].
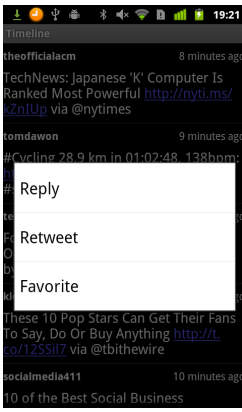
(a) Normal timeline and tweed context menu. (b) Enabling disaster mode. (c) Timeline with high-lighted disaster tweet. (d) Sending tweets in disaster mode.

Figure 2: *Twimight* screenshots.

connection opportunities. The interval has been randomized in order to avoid scanning collisions (i.e., two devices always scanning at the same time will never see each other). Once two phones have discovered one another, they connect to each other and exchange the new disaster tweets, thereby spreading them epidemically. We choose Bluetooth for opportunistic communication, because in spite of its limitations (cumbersome pairing, etc.), it offers an acceptable trade-off between battery lifetime and service provided. Another big benefit is that it can be used without rooting the Android devices. Adding alternative ad hoc communication methods such as WiFi Ad Hoc and WiFi-Opp [8] is planned for the future.

From Disaster to Normal mode: As soon as connectivity to the Twitter servers is re-established, the disaster tweets sent (or re-tweeted) by the user are published to the Twitter server. Publishing only own tweets avoids duplicate publication and authorization issues.

To ensure the reliable dissemination of tweets to the greater number, epidemic spreading (taking every opportunity to replicate a tweet) is mandatory at first. However, with the epidemic dissemination of tweets, not only do we potentially use a lot of network resources but victims of a disaster can also get overwhelmed with the number of received tweets that are displayed in their timeline. There are hence clear needs to scale such dissemination both at the network and human levels. In future work, we consider to limit the spreading of tweets to a given geographical range e.g., 5 km around the source. We will also introduce the new concept of human-triggered re-dissemination where users could retweet a given tweet thus extending its range further e.g., for another 5 km. This would allow important tweets to be spread further. Future work will also consider more efficient and scalable spreading schemes for private messages between users. We intend to leverage opportunistic routing protocols that avoid flooding all the network to reach the destination [9]. Finally, we also plan to integrate resource management strategies to remove tweets as the buffers get full.

3. TWIMIGHT SECURITY EXTENSION

Regular communication through the Twitter service provides both message authenticity and confidentiality. Only authenticated users can submit tweets and private messages between two users are only shown to the intended recipient. Flooding attacks are blocked at the Twitter server by throttling traffic from the same source. We propose a complementary security server to offer comparable security also in the disaster mode where devices exchange tweets among themselves and cannot rely on the online Twitter servers or a centralized security service. We implemented security features to achieve comparable security and to protect *Twimight* at least from basic attacks.

3.1 Assumptions and Requirements

The security architecture of *Twimight* relies on the basic assumption that security elements (certificates, keys, etc.) can be established before a disaster happens. However, once connectivity is lost and the disaster mode is running, the protocols have to work in a completely distributed fashion. Hence, we describe here a *hybrid* architecture based on a centralized infrastructure (i.e., our Twitter security/disaster server and the Twitter servers) for *establishing* the security elements, but *operating* without any infrastructure during a disaster. Such a hybrid scheme simplifies the design of the architecture considerably, compared to a fully distributed security architecture. The downside is that it implies that new users cannot join the network during a disaster. We will address this latter issue in future work.

In terms of threats, we assume that the attacker can communicate with peers in range via Bluetooth, i.e., she can read the messages sent to her during a contact and can send arbitrary messages. Further, she can try to attack the communication of the clients with our centralized security infrastructure. However, we assume that the attacker does *not* have access to the phone of benign users (i.e., we trust the normal users' phones, including the operating system and other Apps running on the phone).

(a) Normal operation.
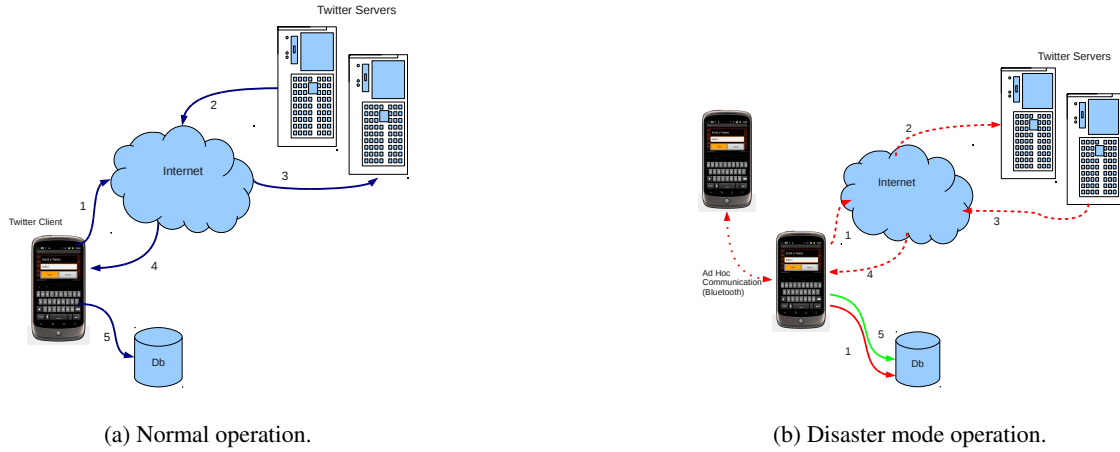
(b) Disaster mode operation.

Figure 3: Two modes of operation. Continuous blue lines mean normal operations. Dashed red lines refer to attempts in disaster mode, whilst continuous red lines means operations that are performed without problems in disaster mode.

Based on these assumptions, we specify the following three requirements for our security architecture:

1. Authenticity of disaster tweets has to be verifiable, to prevent an attacker from lying about the sender of a message.

2. Confidentiality of *direct messages* between Twimight users using encryption in disaster mode such that an attacker receiving such a message for relaying cannot read its content.

3. Anti-spam scheme minimizing the impact of an attacker flooding the network with useless messages (denial of service attack).

For the first two requirements, we use a Public Key Infrastructure (see Section 3.2). For the last requirement, we use a buffer and prioritization strategy which can limit the spreading of spam (see Section 3.5).

3.2 Public Key Infrastructure

To support the implementation of security in Twimight, we introduce the *Twimight Disaster Server (TDS)*. The TDS essentially serves as a *Certificate Authority* with three main functions: 1) Issuing certificates for public keys (thereby binding a public key to a Twitter ID), 2) distributing public keys, and 3) revoking invalid keys. Figure 4 shows an overview of the security architecture. The TDS is the only certificate authority we allow for signing public keys. We distribute the root certificate with the Twimight installation package[3].

Communication with the TDS is encrypted using the HTTPS protocol to prevent eavesdropping. At the TDS, the client is identified by its Twitter user ID, which is sent (along with the other parameters of a request) by the HTTP POST method. To guarantee the authenticity of the request, the client also

sends the request_token which it obtained in the OAuth authorization process of the client with the Twitter server. The Twitter user ID together with this token is then used by the TDS to query the Twitter server to verify the identity. The request_token is cached at the TDS for the duration of its validity for authentication of further requests without querying the Twitter API each time[4].

In the following, we describe the key management, from generation to revocation of certificates and their associated public/private key pairs.

Key generation: Twimight creates public/private key pairs locally on the handset to avoid sending private keys over the network. Twimight checks whether a new key pair has to be generated: 1) Upon startup of the application and after logging in to Twitter, 2) Periodically after $T = 7$ days (the reason for this is explained below, see "Key Expiration Date").

If such a check does not find a valid key pair (i.e., no key at all, no certificate or an outdated key), Twimight generates a new key pair.

Key certification: After creation of the key pair, the client sends the public key to the TDS for certification. The certificate – including the serial number, public key, Twitter ID and expiration date – is sent back to the client. Upon reception, the client verifies the signature with the root certificate. If this check fails or no response is received from the TDS, the client starts over with generating a new key pair. It makes 3 attempts before giving up and waiting for the next key generation check to try again. Note that the option to turn on the disaster mode is greyed out in the settings until a key is successfully signed. Further, to prevent denial of service attacks, only one certificate per user per day is issued.

Key storage: On the client side, the private key as well as the public key and the corresponding certificate are stored as

---

[3]Updating the root certificate requires an update of the application.

[4]Note that this token will also allow us posting (signed) disaster tweets on behalf of a user from the TDS to Twitter. But that's not part of the security architecture.
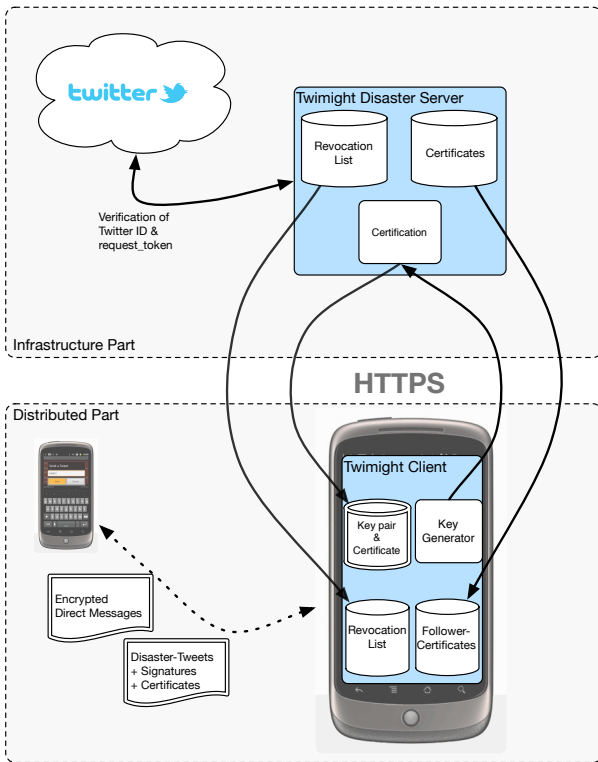
Figure 4: Hybrid (infrastructure / distributed) security architecture.

an Android *shared preference* to make them inaccessible to other applications on the phone. On the server side, during the key certification requested by a client, the TDS stores the public key and the corresponding certificate with its expiry date in its certificates database.

Explicit Key Revocation To invalidate keys (e.g. stolen phone) the TDS maintains a revocation list. Normally, clients poll the certificate authority using the Online Certificate Status Protocol (OCSP) during key validation to check its validity. Since this online check cannot be performed by the clients during a disaster, we cannot rely on OCSP and have to cache the revocation list locally on the devices. To keep the local copy up-to-date, the Twimight application synchronizes the revocation list during normal operations after startup or periodically after one day.

In case a phone gets stolen, to actually perform key revocation, the user has to (re)login (on a different device) and trigger the key revocation from the application settings. Further, we plan to implement key revocation service provided by a web application hosted on the TDS to allow revocations after a successful authentication.

Key Expiration Date The revocation list contains all revoked keys that are not yet expired. Since we have to distribute the revocation list to mobile phones, we are interested in limiting its size to save bandwidth and power. Therefore, the keys signed by the TDS have a lifetime of only 14 days, which is rather short compared to typical certificate lifetimes of several years. This ensures that entries on the revocation

list can be removed timely and keeps the revocation list compact.

To ensure that all clients have a valid certificate during a disaster, they have to update their keys before they expire. In more detail, we propose that clients already update their key after *half* of its lifetime. This enforces that all Twimight users have a valid key for at least the next $T = 7$ days. Please note that we currently assume that after 7 days some network infrastructure (permanent or temporary) should be up again. Further, the key lifetime can easily be adjusted anytime to optimize this trade-off between the length of the revocation list and the expected disaster duration. In addition, this approach allows us to adjust the key life time for individual users. For example, we plan to distribute keys with extended lifetime to special Twimight users like official rescue teams.

3.3 Disaster Tweet Authenticity

Once in disaster mode, Twimight applications are scanning for new peers to exchange disaster tweets. These disaster tweets include the message text and meta-data such as creation time and tweet ID. This data is signed (using the user's private key), the user's certificate is attached (public key) and the message is stored in the local disaster database together with the previously received tweets. See Figure 7 for the detailed format.

Before transmitting, and after receiving, each peer performs the following checks to validate the tweets:

1. The creation time of a tweet must be in the past [5] .

2. The certificate's validity is checked with the root certificate embedded in Twimight.

3. The certificate is checked for validity against the local revocation list and the expiration date is controlled.

4. The tweet signature is verified.

The tweet's signature allows checking the integrity of the text and meta-data, preventing any attacker from tampering for example with the creation time. It also allows authenticating the tweet's author. In any case, if a check fails the tweet is discarded. All valid tweets are entered in the `DisasterTable` buffer, ordered by their creation time.

3.4 Direct Message Confidentiality

Direct messages are a way of communicating privately in Twitter. To guarantee privacy even if direct messages are forwarded from device to device, they are encrypted with the public key of the receiver. Hence, we must make sure to have all relevant public keys readily available once communication breaks. Since Twitter only allows sending direct messages to the followers of a user, this is a manageable task: At startup time and periodically after one day of runtime, Twimight polls the TDS to provide the active public keys of all the followers.

---

[5]To account for small deviations of the clocks, we allow the creation time to be at most $\Delta t = 1h$ in the future.

The user interface prevents sending direct messages to users for which no public key is available (i.e., if the user attempts to enter a direct message she is notified about the lack of key). The content of a direct message is encrypted using the destination's public key. The tweet and its meta-data is then signed as described above in 3.3 to provide authenticity of the sender and data integrity. The sender's certificate is appended and the message is stored in the local disaster database. For details about the packet format, see Figure 7.

During peering, the direct messages are exchanged. Before transmitting, and after receiving, the peers perform the following checks:

1. The creation time of a tweet must be in the past [5].

2. The certificate's validity is checked with the root certificate embedded in Twimight.

3. The certificate is checked for validity against the local revocation list and the expiration date is controlled.

4. The tweet signature is verified.

If one of these tests fails, the message is discarded, otherwise the message is inserted into the `DisasterDMTable` buffer. Further, if a node is the destination of a direct message, the message is decrypted and the user is notified.

3.5 Spam Control

It is known that user behavior and in particular tweet volume of Twitter user varies broadly from few tweets a month to multiple tweets per hour. In order to prevent (too) high volume users ("spammers") from claiming the full network resources, we propose to rely on two simple mechanisms. In a first step, we ask the users to voluntarily limit their tweeting to important news by displaying a friendly warning to the user after each 20 disaster tweets. Recall that we rely on user collaboration and self-organization in a disaster, and hence believe that simply raising user awareness is effective.

As a second, more technical solution, we implement the following constraints for message exchange: 1) We limit the number of messages which are sent from one peer to the other to $M_{tot} = 500$. Since the maximum size of a message is known (140 characters + meta-data of fixed length), we can drop communication if a peer wants to send us more than the allowed amount of data. 2) Out of the M_{tot} tweets transmitted, we reserve a maximum of $M_{self} = 250$ spots for *our own tweets (and re-tweets)*.

This ensures that even in presence of high volume users we can maintain at least $\frac{M_{self}}{M_{tot}}$ of the capacity for other communication. This splitting is important since we assume that users start heavily re-tweeting content that they consider important to fill the reserved capacity with useful content. This human-based filtering could be supported by filtering functions in the user interface which for example allow to display only re-tweets, or to filter out the tweets of a certain user.
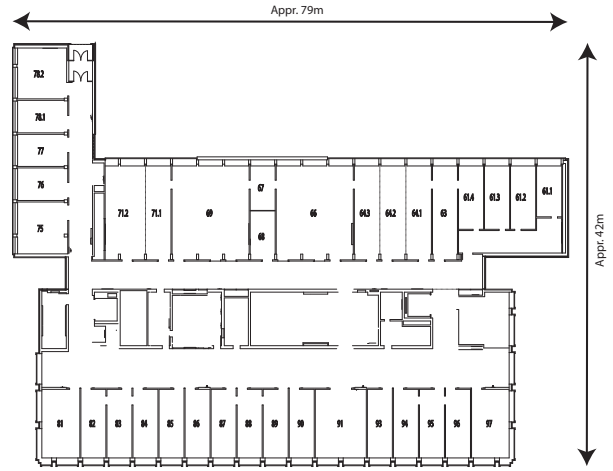

Figure 5: G-floor layout in the ETZ building at ETH Zurich.

4. VALIDATION

As a first step towards validating and evaluating Twimight, we have carried out multiple small-scale experiments in our building at ETH Zurich focusing on the basic opportunistic functionality[6]. The mobility patterns and the interactions reflect a typical office scenario. The experiments involved five people working on two floors in the ETZ building (see Figure 5), who were carrying Nexus One phones running Android 2.3.3 during usual working hours. The longest experiment lasted 2 days and included 915 tweets. To generate a good amount of traffic we have used an automatic tweet generator. This implies that a part of the tweets were generated during the night, when users were inactive (at home). To account for these circumstance we have identified and separated them.

4.1 Delivery Delay

The delivery delay is an important metric used to evaluate opportunistic network and routing protocols performance. It measures the time between creation and delivery.

Figure 6a represents the delivery delay CDF of Tweets (automatic and manually generated). This illustrates that around 30% of the messages created during the day were delivered within 4 minutes. This can be attributed to the office scenario, where office mates are almost constantly connected. Interestingly, after about 2 hours, almost all messages were delivered, which shows the efficiency of the epidemic spreading.

4.2 Hop Count

Another interesting parameter used to evaluate opportunistic networks is the hop count distribution. Results are shown in Figure 6b: 49% of the Tweets were delivered directly while 51% tweets were relayed over multiple hops. This

---

[6]The experiments for evaluating the security architecture and features are under way.

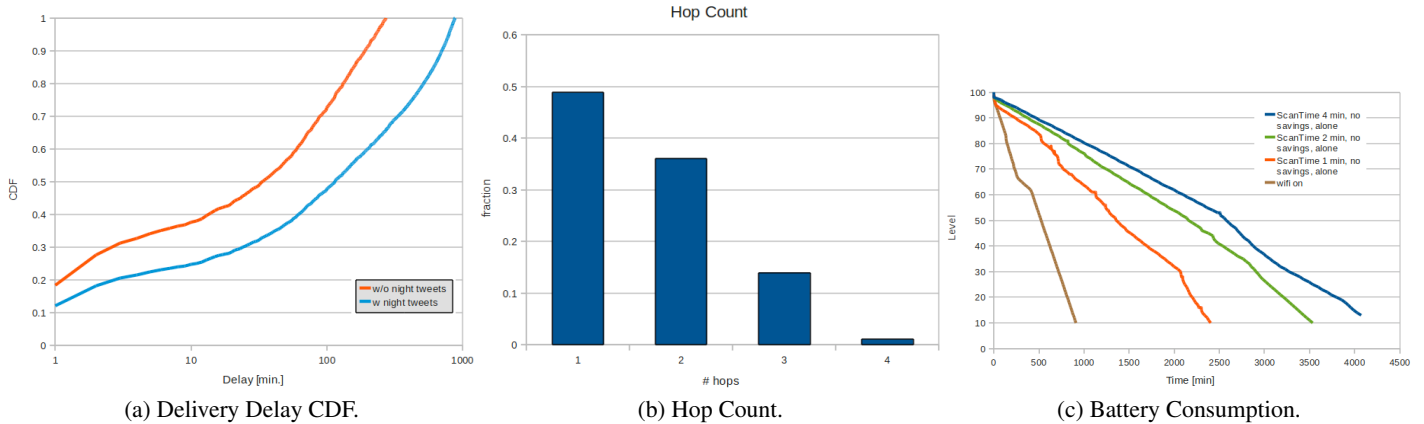| (a) Delivery Delay CDF. | (b) Hop Count. | (c) Battery Consumption. |

Figure 6: Performances of Twimight (w/o security features).

shows that Twimight provides a reliable multi-hop dissemination service.

4.3 Power consumption

Since mobile devices are constrained on battery power (especially in a disaster where the power infrastructure may be damaged), a critical parameter for Twimight is energy efficiency. Hence, we evaluated the power consumption of different Bluetooth scanning intervals (1, 2 and 4 minutes) as well as WLAN ad hoc communication, by sampling the battery level periodically. To limit the impact of external influences on the outcome, we ensured that no phone calls or text messages were sent or received during the experiment (even though the phones were connected to the GSM network). The display was only turned on a few times during the measurement.

In Figure 6c we present the battery level over time for a single measurement. We observe that the highest energy consumption is caused WiFi Ad Hoc communication. In this case, the battery lasted only for 16 hours. Better results were achieved with using Bluetooth. Clearly, the scanning interval affects the uptime strongly. A scanning interval of 1 minute resulted in an uptime of 41 hours. Increasing the interval to 2 minutes extends the lifetime significantly by a factor of almost 1.5 to 58 hours. Further increase of the scanning interval to 4 minutes results in 68 hours uptime.

Based on these results, we have chosen a scanning interval of 2 minutes a good trade-off in terms of battery lifetime and missing short forwarding opportunities.

5. RELATED WORK

Public key cryptography is a widely used convention to uniquely authenticate different entities based on certificates issued by a trusted Certificate Authority (CA) or CA hierarchy. In an opportunistic or ad hoc network, the CA duty can be handed over to nodes which have to sign certificates of others once secure pairing is performed. The crucial part then comprises the dissemination of the certificates in a distributed way in order for nodes to build user-based certificate chains. In [10] and [11], every entity generates its own self-created credentials, lets it be signed by other entities and distributes it to nodes it encounters. While in [10] nodes can build certificate chains similar to PGP [12], in [11] the signature process is only allowed through secure pairing or over a common friend. The problem with both approaches is the fact that certificate chains may break in opportunistic networks and conscious signing may lead to problems when meeting complete strangers of which no prior information, connection or other reason to trust is available. In this paper, we considered a hybrid approach where we assumed users can access the PKI infrastructure (our Twimight Disaster server) before a disaster occurs. This simplifies greatly the management of certificates.

Temporary deployments of wireless mesh networks [13] have been proposed for disaster relief. However, the nodes for building a wireless mesh networks have to be distributed first and then deployed. This requires access of an emergency response team to the disaster area. We can imagine that such an operation, even if the emergency response forces are well equipped and trained, takes at least a few hours (in case of difficult terrain maybe even days) to execute. Also, it is difficult to get a complete coverage without careful planning. Hence, during these first hours after a disaster – the most critical time in which many lives can be saved – people are left on their own, without any means of communication to organize themselves. Using satellite communications, suffers from the same setup time since satellite phones are not usually readily available to regular people in disasters.

6. CONCLUSIONS

In this paper, we presented Twimight, a Twitter application for Android phones relying on opportunistic communications to spread tweets in an epidemic fashion in disaster situations. Moving away from a online centralized service to a distributed and delay-tolerant service relying on oppor-

tunistic communications required extending the security architecture of Twitter for Twimight. We proposed security extensions to allow (i) tweets being relayed over opportunistic hops in a secure fashion (integrity, authentication), (ii) the confidentiality of direct tweet messages, and (iii) a simple anti-spam scheme avoiding misbehaving users to flood the network with less important tweets.

Acknowledgment

7. REFERENCES

[1] A. Chowdhury, "Global pulse," http://blog.twitter.com/2011/06/global-pulse.html, 2011.

[2] A. Bruns, "Tracking crises on twitter: Analysing #qldfloods and #eqnz." Presented at the Emergency Media and Public Affairs Conference, Canberra, 2011.

[3] A. Vahdat and D. Becker, "Epidemic routing for partially-connected ad hoc networks," Tech. Rep., 2000.

[4] W. Zhao, M. Ammar, and E. Zegura, "A message ferrying approach for data delivery in sparse mobile ad hoc networks," in *ACM MobiHoc*, 2004.

[5] E. Nordström, P. Gunningberg, and C. Rohner, "A search-based network architecture for content-oriented and opportunistic communication," *Uppsala University Technical Report 2009-003*, 2009, http://code.google.com/p/haggle.

[6] PodNet Project. [Online]. Available: http://www.podnet.ee.ethz.ch/

[7] T. Hossmann, F. Legendre, P. Carta, P. Gunningberg, and C. Rohner, "Twitter in disaster mode: Opportunistic communication and distribution of sensor data in emergencies," in *ExtremeCom*, 2011.

[8] S. Trifunovic, B. Distl, D. Schatzmann, and F. Legendre, "Wifi-opp: Ad-hoc-less opportunistic networking," in *ACM CHANTS*, 2011.

[9] T. Hossmann, T. Spyropoulos, and F. Legendre, "Know thy neighbor: Towards optimal mapping of contacts to social graphs for DTN routing," in *IEEE Infocom*, 2010.

[10] J. P. H. S. Capkun, L. Buttyan, "Self-organized public-key management for mobile ad-hoc networks," in *IEEE Transactions on Mobile Computing*, 2003.

[11] L. B. S. Capkun, J. P. Hubaux, "Mobility helps peer-to-peer security," in *IEEE Transactions on Mobile Computing*, 2006.

[12] P. R. Zimmermann, *The Official PGP User's Guide*, MIT press, 1995.

[13] R. Bruno, M. Conti, and E. Gregori, "Mesh networks: commodity multihop ad hoc networks," *Communications Magazine, IEEE*, 2005.
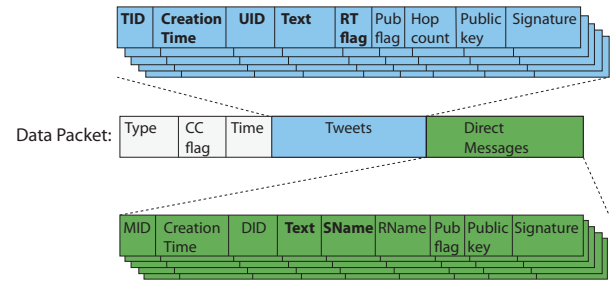
APPENDIX - Twimight Packet Format

Figure 7: Data packet format. Bold fields are encrypted in Direct Messages respectively signed in Tweets.

Disaster Tweets: The format of disaster tweets is shown at the top of Figure 7. The data fields shown in the figure are summarized in Table 1. The signature is calculated over the fields TID, Creation Time, UID, Text and RT flag.

Field	Purpose
TID	Tweet ID, assigned by sender node.
Creation Time	Timestamp of sending the tweet.
UID	Twitter user ID of sender.
Text	Tweet text.
RT flag	Mark re-tweets.
Pub flag	Mark disaster tweets already published to Twitter.
Hop count	Nr. of forwarded hops.
Public key	Signed public key of sender.
Signature	Digital signature of tweet.

Table 1: Data fields of a disaster tweet.

Direct Messages: The format of direct messages is shown at the bottom of Figure 7. The fields are summarized in Table 2.

Field	Purpose
MID	Direct message ID, assigned by sender node.
Creation Time	Timestamp of sending the tweet.
DID	Twitter ID of destination user.
Text	Direct message text.
SName	Twitter username of sender.
RName	Twitter username of receiver.
Pub flag	Mark disaster tweets already published to Twitter.
Public key	Signed public key of sender.
Signature	Digital signature of tweet.

Table 2: Data fields of a disaster direct message.