SWEET: Serving the Web by Exploiting Email Tunnels

Wenxuan Zhou[1] Amir Houmansadr[2] Matthew Caesar[1] Nikita Borisov[1]

[1]University of Illinois at Urbana-Champaign, [2]University of Texas at Austin

Abstract. Open communication over the Internet poses a serious threat to countries with repressive regimes, leading them to develop and deploy censorship mechanisms within their networks. Unfortunately, existing censorship circumvention systems do not provide high *availability* guarantees to their users, as censors can easily identify, hence disrupt, the traffic belonging to these systems using today's advanced censorship technologies. In this paper we propose SWEET, a highly available censorship-resistant infrastructure. SWEET works by encapsulating a censored user's traffic inside email messages that are carried over by typical email service providers, like Gmail and Yahoo Mail. As the operation of SWEET is not bound to any specific email provider we argue that a censor will need to block all email communications in order to disrupt SWEET, which is unlikely as email constitutes an important part of today's Internet. Through experiments with a prototype of our system we find that SWEET's performance is sufficient for web traffic. In particular, regular websites are downloaded within couple of seconds.

Keywords: Censorship circumvention; email communications; traffic encapsulation

1 Introduction

The Internet provides users from around the world with an environment to freely communicate, exchange ideas and information. However, free communication continues to threaten repressive regimes, as the open circulation of information and speech among their citizens can pose serious threats to their existence. As a result, repressive regimes extensively monitor their citizens' access to the Internet and restrict open access to public networks [37] by using different technologies, ranging from simple IP address blocking and DNS hijacking to the more complicated and resource-intensive Deep Packet Inspection (DPI) [3, 22].

With the use of censorship technologies, a number of different systems were developed to retain the openness of the Internet for the users living under repressive regimes [2, 5, 9, 10, 12, 19]. While these circumvention tools have helped, they face several challenges. We believe that the biggest one is their lack of *availability*, meaning that a censor can disrupt their service frequently or even disable them completely [14, 24, 26, 27, 31]. The common reason is that the network traffic made by these systems can be distinguished from regular Internet traffic by

censors, i.e., such systems are not *unobservable*. To improve availability, recent proposals for circumvention aim to make their traffic unobservable to the censors by pre-sharing secrets with their clients [6, 11, 13]. Others [16, 18, 21, 36] suggest to conceal circumvention by making infrastructure modifications to the Internet. Nevertheless, deploying and scaling these systems is a challenging problem.

A more recent approach in designing unobservalbe circumvention systems is to imitate popular applications like Skype and HTTP, as suggested by SkypeMorph [28], CensorSpoofer [34], and StegoTorus [35]. However, it has recently been shown [15] that these systems' unobservability is breakable; this is because a comprehensive imitation of today's complex protocols is sophisticated and infeasible in many cases. A promising alternative suggested [15, 17] is to not mimic protocols, but run the actual protocols and find clever ways to tunnel the hidden content into their genuine traffic; this is the main motivation of the approach taken in this paper.

In this paper, we design and implement SWEET, a censorship circumvention system that provides high availability by leveraging the openness of email communications. Figure 1 shows the main architecture. A SWEET client, confined by a censoring ISP, tunnels its network traffic inside a series of email messages that are exchanged between herself and an email server operated by SWEET's server. The SWEET server acts as an Internet proxy [23] by proxying the encapsulated traffic to the requested blocked destinations. The SWEET client uses an oblivious, public mail provider (e.g., Gmail, Hotmail, etc.) to exchange the encapsulating emails, rendering standard email filtering mechanisms ineffective in identifying/blocking SWEET-related emails. There are two projects that work in a similar manner to SWEET: FOE [1] and MailMyWeb [25]. Instead of tunneling traffic as in SWEET, these systems simply download a requested website and send it as an email attachment to the requesting user. This highly limits their performance, as users can only access static websites.

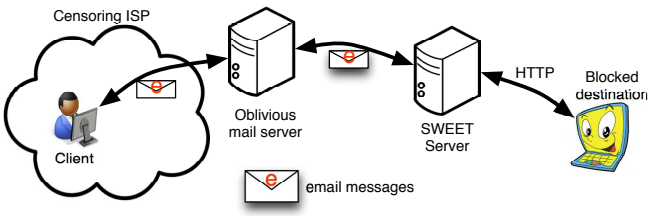

Fig. 1. Overall architecture of SWEET.

SWEET's unobservability We claim that a censor is not easily able to distinguish between SWEET's email messages and benign email messages. As described later in Section 3, a SWEET client has two options in choosing her email account : 1)*AlienMail* a non-domestic email that encrypts emails (e.g., Gmail for users in China), and 2)*DomesticMail* a domestic email account without encryp-

tion (e.g., 163.com for users in China). When AlienMail is used, all of SWEET emails are sent to a publicly known email address, e.g., `tunnel@sweet.org`, encrypted; however, a censor will not be able to identify these emails since they are *proxied* by the AlienMail server running outside the censoring area. In other words, the censor only observes that the client is exchanging encrypted messages with the AlienMail server (e.g., Gmail's mail server in U.S.), but he will not be able to observe neither the recipient's email address, nor the IP address of the sweet.org mail server. As a result, **existing approaches for spam filtering such as shooting the spamming SMTP servers and dropping spam emails are entirely infeasible**. In the case of DomesticMail, the SWEET server uses a secondary *secret* email account, which is only shared with that particular client, for exchanging SWEET emails (i.e., `myotheremail@163.com` instead of `tunnel@sweet.org`). Thus, the censor will not be able to identify SWEET messages from their recipient fields (since the censor does not know the association of `myotheremail@163.com` with SWEET). Also, the use of steganography/encryption to embed tunneled data renders DPI infeasible.

SWEET's availability Given SWEET's unobservability discussed above, a censor can not efficiently distinguish between SWEET emails and benign email messages. Hence, in order to block SWEET a censor needs to block all email messages to the outside world. However, email is an essential service in today's Internet and it is very unlikely that a censorship authority will block *all* email communications to the outside world, due to different financial and political reasons. This, along the fact that SWEET can be reached through a wide range of domestic/non-domestic email providers provides a high degree of *availability* for SWEET.

In fact, the high availability of SWEET comes for the price of higher, but bearable, communication latencies. Figure 2 compares SWEET with several popular circumvention systems regarding their availability and communication latency. As our measurements in Section 5 show, SWEET provides communication latencies that are convenient for latency-sensitive activities like web browsing (i.e., few seconds).

In summary, this paper makes the following main contributions: i) we propose a novel infrastructure for censorship circumvention, SWEET, which provides high availability, a feature missing in existing circumvention systems; ii) we develop two prototype implementations for SWEET (one using webmail and the other using email exchange protocols) that allow the use of nearly all email providers by SWEET clients; and, iii) we show the feasibility of SWEET for practical censorship circumvention by measuring the communication latency of SWEET for web browsing using our prototype implementation.

The rest of this paper is organized as follows, in Section 2, we reviews our threat model. We provide the detailed description of the proposed system, SWEET, in Section 3. We present our prototype implementation and evaluations in Sections 4 and 5, respectively, and conclude in Section 6.

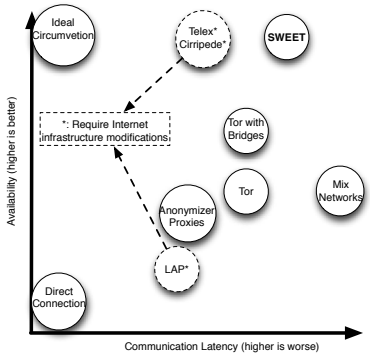

Fig. 2. Availability and communication latency comparison of circumvention systems.

2 Threat model

We assume that a user is confined inside a censoring ISP. The ISP blocks the user's access to certain Internet destinations. The censor is assumed to be able to perform passively monitoring, for instance, using deep packet inspection techniques [22], and also to actively manipulate its traffic, by selectively dropping packets, and adding latency to some packets, to disrupt the use of circumvention systems and/or to detect the users of such systems.

We assume that the censorship is constrained not to degrade the *usability* of the Internet. In other words, even though it *selectively* blocks certain Internet connections, she is not willing to block key Internet services *entirely*. In particular, the operation of SWEET system relies on the fact that a censoring ISP does not block *all* email communications, even though she can selectively block emails/email providers. We also assume that the ISP has as much information about SWEET as any SWEET client.

3 Design of SWEET

In this section, we describe the design of SWEET. Figure 1 shows the overall architecture. SWEET tunnels network connections between a client and a server inside email communications. Upon receiving the tunneled network packets, the SWEET server acts as a transparent proxy between the client and the network destinations requested by the client.

A client's choices of email services

i) AlienMail An AlienMail is a mail provider whose mail servers reside outside the censoring ISP, e.g., Gmail for the Chinese clients. We only consider AlienMails that provide email encryption, e.g., Gmail and Hushmail. A SWEET client who uses an AlienMail does not need to apply any additional encryption/steganography to her encapsulated contents. She simply sends her emails to

the publicly advertised email address of SWEET server, e.g., `tunnel@sweet.org`, since the censors will not be able to observe (and block) the address, which is in an encrypted format.

ii) DomesticMail A DomesticMail is an email provider hosted inside the censoring ISP and possibly collaborating with the censors, e.g., 163.com for the Chinese clients. Since the censors are able to observe the email contents, the SWEET client using a DomesticMail should hide the encapsulated contents through steganography (e.g., by doing image/text steganography inside email messages). Also, the client can not send her SWEET emails to the public email address of SWEET server since the mail recipient is observable to the DomesticMail provider and/or the censor. Instead, the client generates a secondary email address, `myotheremail@somedomain.com`, and then provides the email credentials for this secondary account to SWEET server through an out-of-band channel. The SWEET server uses this email address to exchange SWEET emails *only* with this particular client.

In the following, we describe the details of SWEET's server and client architectures. Without loss of generality, we **only consider the case of AlienMail**. If DomesticMail is used, the client and server should also perform some steganography operations to hide the encapsulated traffic, as well as they should exchange a secondary email address.

3.1 SWEET server

The SWEET server is running outside the censoring region. It helps SWEET clients to evade censorship by proxying their traffic to blocked destinations. Figure 3 shows the design, composed of four elements: Email agent, Converter, Proxy agent, and Registration agent. Here the Email agent is an IMAP and SMTP server.

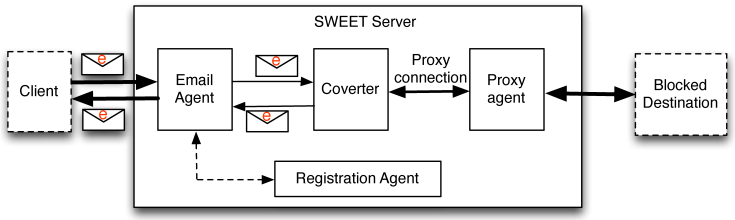

Fig. 3. The main architecture of SWEET server.

The email agent of the SWEET server receives two type of emails; *traffic emails*, containing tunneled traffic from the clients (sent to `tunnel@sweet.org`), and *registration emails*, which carry client registration information (to `register@sweet.org`).

Client registration: Before the very first use of the SWEET service, a client needs to register her email address with the system. This is automatically performed by the client's SWEET software.The objective of client registration is twofold: to prevent denial-of-service (DoS) attacks and to share a secret key between a client and the server. A DoS attack might be launched on the server to disrupt its availability, e.g., through sending many malformed emails on behalf of non-existing email addresses. In order to register (or update) the email address of a client, the client's SWEET software sends a registration email from the user's email address, to the SWEET's registration email address. The email agent forwards registration emails to the registration agent. For any new registration request, the registration agent generates and sends an email to the requesting email address (through the email agent) that contains a unique computational *challenge* (e.g., [20]). After solving the challenge, the client software sends a second email to `register@sweet.org` that contains the solution, along with a Diffie-Hellman [32] public key $K_C = g^{k_C}$. If the client's response is verified by the registration agent, the client's email address will be added to a *registration list*, which contains registered email addresses with their expiration time. Also, the registration agent uses its own Diffie-Hellman public key, $K_R = g^{k_R}$, to evaluate a shared key $k_{C,R} = g^{k_R k_C}$ for the later communications with the client. The registration agent adds this key to the client's entry in the registration list. The client is able to generate the same $k_{C,R}$ key using SWEET's publicly advertised public key and her own private key [32].

Tunneling the traffic: Any traffic email received by the email agent is processed as follows: the email agent forwards the email to the converter. The converter processes extracts the tunneled information. The converter, then, decrypts the information (using the key $k_{C,R}$ corresponding to the user) and sends it to the proxy agent. Finally, the proxy sends it to the requested destination. Similarly, for any tunneled packet received from the destinations, the proxy agent sends it to the converter. The converter encrypts the received packet(s), and generate a traffic email with the encrypted data as an attachment, targeted to the email address of the corresponding client. The generated email is passed to the email agent, who sends the email to the client. Note that to improve the latency performance, small packets that arrive at the same time get attached to the same email.

3.2 SWEET client

To use SWEET, a client needs to obtain a copy of SWEET's client software and install it on her machine. The client also needs to create one email account, and to configure the SWEET's software with information of her email account. Prior to the first use, the client software registers the email address of its user with the SWEET server and obtains a shared secret key $k_{C,R}$.

We propose two designs for SWEET client: a protocol-based design, which uses standard email protocols to exchange email with client's email provider, and a webmail-based design, which uses the webmail interface of the email provider. We describe these two designs in the following.

Protocol-based design Figure 4a) shows the three main elements.
❶ **Web Browser:** The client can use any web browser that supports proxying of connections, e.g., Google Chrome, Internet Explorer, or Mozilla Firefox. The client needs to configure her browser to use a local proxy server. The client can use two different browsers for browsing with and without SWEET to avoid the need for frequent re-configurations of the browser. Alternatively, some browsers (e.g., Chrome, and Mozilla Firefox) allow a user to have multiple browsing *profile*s, hence, a user can setup two profiles for browsing with and without SWEET.
❷ **Email Agent:** It sends and receives SWEET emails thorough the client's email account. It is configured with the settings of the SMTP and IMAP/POP3 servers of the user's email account, as well as the login information.
❸ **Converter:** It sits between the web browser and the email agent, and converts SWEET emails into network packets and vice versa. It uses the keys shared with SWEET, $k_{C,R}$, to encrypt/decrypt email content.

Once the client enters a URL into the configured browser (❶), the browser makes a proxy connection to the local port that the converter (❸) is listening on. The converter accepts the proxy connection and keeps the state of the established TCP/IP connections. For packets that are received from the browser, the converter generates traffic emails, to `tunnel@sweet.org`, having the received packets as encrypted email attachments (using the key $k_{C,R}$). Such emails are passed to the email agent (❷), which sends the emails to the SWEET server.

The email agent also continuously looks for new emails from the server. Once new emails are received, the email agent passes them to the converter, who in turn extracts the packets from the emails, decrypts them, and sends them to the browser over the existing TCP/IP connection.

Webmail-based design Alternatively, the SWEET client can use the webmail interface of the client's public email provider. as showed in Figure 4b).The main difference with the protocol-based design is that in this case the email agent (❷) uses a web browser to exchange emails. More specifically, the email agent uses its web browser to open a webmail interface with the client's email account, using the user's authentication credentials for logging in. Through this HTTP/HTTPS connection, the email agent communicates with the SWEET server by sending and receiving emails.

4 Prototype Implementation

4.1 Server implementation

We implement the SWEET server on a Linux machine, which runs *Ubuntu 10.04 LTS* and has a 2 GHz quad-core CPU and 4 GB of memory. We run Postfix[1], a simple email server that supports basic functions. Postfix listens for new emails targeted to the `register@sweet.org` and `tunnel@sweet.org`. Postfix stores the

[1] http://www.postfix.org/

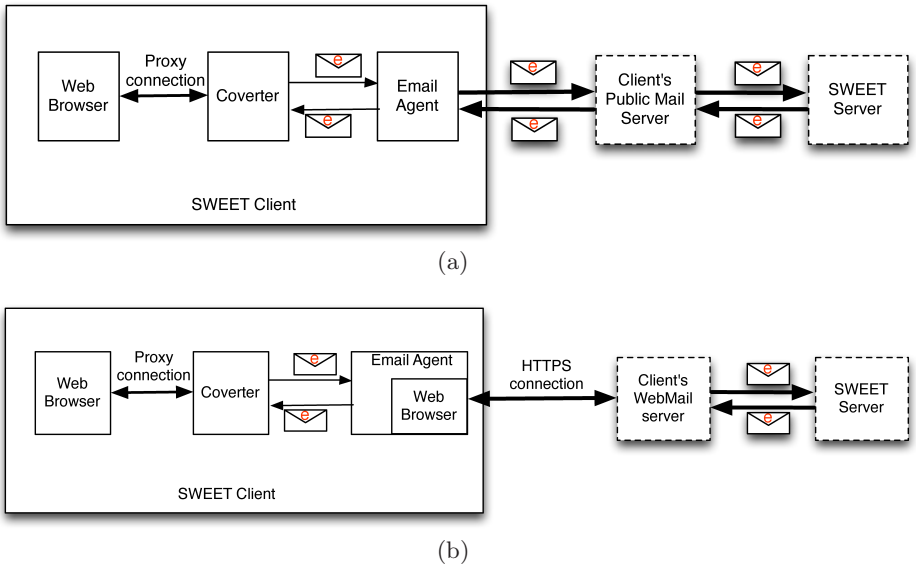

Fig. 4. (a) The protocol-based and (b) the webmail-based design for SWEET client.

received emails into designated file directories that are continuously watched by the converter and registration agent of SWEET server. Each stored email has a unique name, concatenating the email id of its corresponding client and an increasing counter. The converter agent is a simple Python-based program that runs in the background and continuously checks the folder for new emails. The converter also converts proxied packets, passed by SWEET's proxy, into emails and sends them to their intended clients. For the proxy agent, we use Squid[2] as our HTTP proxy and Suttree[3] as our SOCKS proxy. Each listens on a local port for connections from the converter.

4.2 Client implementation

We implement both protocol-based and webmail-based versions of the SWEET client on a desktop machine, running *Linux Ubuntu 10.04 TLS*.

Protocol-based design We set up a web browser to use the local port "localhost:9034" as the SOCKS/HTTP proxy. The converter is a simple python script that listens on port 9034 for connections, e.g., from our web browser. We implement the email agent of SWEET client using *Fetchmail*[4], a popular client software for sending and retrieval of emails through email protocols. We generate a free Gmail account and configure Fetchmail to receive emails through IMAP

[2] http://www.squid-cache.org/
[3] http://suttree.com/code/proxy/
[4] http://www.fetchmail.info/

and POP3 servers of Gmail, and to send emails through the SMTP server of Gmail. Note that our design does not rely on Gmail, and the client software can be set up with any email account.

Webmail-based design Our webmail-based implementation uses the same converter as the one used in the protocol-based prototype. A Google Chrome browser is used for making connections through SWEET, configured to use "localhost:9034" as a proxy. We prototype the web-based email agent by running a *UserScript*[5] inside the Mozilla Firefox browser. More specifically, we install a Firefox extension, *Greasemonkey*[6], to allow a user to run her own JavaScript, i.e., Userscript, while browsing certain destinations. We write a UserScript that runs in Gmail's webmail interface and listens for the receipt of new emails. Our UserScript saves new emails in a local directory, which is watched by the converter. Note that the Firefox browser is directly connected to the Internet and does not use any proxies (user needs to use the configured Chrome browser to surf the web through SWEET).

5 Evaluation

We evaluate SWEET using our prototype implementation.

5.1 Performance

We use Gmail as the mail provider in our experiments. Our SWEET server is located in Urbana, IL, resulting in approximately 2000 miles of geographic distance between the SWEET server and Gmail's email server (we locate Gmail's location from its IP address). Figure 5a) shows the CDF of the time that a SWEET email sent by a client takes to reach our SWEET server (the reverse path takes a similar time). As the figure shows, around 90% of emails take less than 3 seconds, which is very promising considering the high data capacity of these emails. Note that based on our measurements, most of this delay comes from email handling (e.g, spam checks, making SMTP connections, etc.) performed by the mail provider , but not from the network latency. As a result, the latency would be very similar for users with an even longer geographical distance from the mail server.

Client registration Before being able to request data from Internet destinations, a user needs to be registered by the SWEET server. Figure 5b) shows the time taken to exchange registration messages between a client and the SWEET server. Note that the client registration needs to be performed *only once* for a long period of time. The figure shows that more than 90% of registrations establish in less than 8 seconds (with an average of 6.4 seconds).

We use two metrics to evaluate the latency performance of SWEET in browsing websites: the *time to the first appearance (TFA)* and the *total browsing time*

[5] http://userscripts.org/
[6] https://addons.mozilla.org/en-US/firefox/addon/greasemonkey/

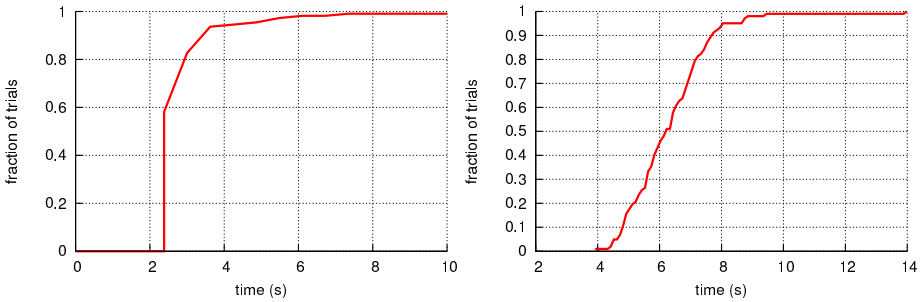

Fig. 5. The CDF of (a) the time that a SWEET email takes to travel from the SWEET client to the SWEET server; (b) the registration time.

(TBT). The TFA is the time taken to receive the first response from a requested destination. It is an important metric in measuring user convenience during web browsing. For instance, suppose that a client requests a URL, e.g., `http://www.cnn.com/some_news.html`. By the TFA time the client receives the first HTTP RESPONSE(s), which include the URL's text parts (perhaps the news article) along with the URLs of other objects on that page, e.g., images, ads hosted by other websites, etc. At this time the client can start reading the received portion of the website, while her browser sends requests for other objects on that webpage. On the other hand, the total browsing time (TBT) is the time after which the browser finishes fetching all of the objects in the requested URL.

Using our prototype we measure the end-to-end web browsing latency for the client to reach different web destinations. Figure 6a) shows the TFA for the top 10 web URLs from Alexa's most-visited sites ranking [4]. The median is about 5 seconds across all experiments, which is very promising to user convenience.

On the other hand, Figure 6b) shows the TBT for the same set of destinations (50 runs for each). As can be seen, the destinations that contain more web objects (e.g., yahoo and linkedin) take more time to get completely fetched (note that after the TFA time the user can start reading the webpage).We also run similar experiments through the popular Tor [33] anonymous network. Figure 7 compares the latency CDF for SWEET and Tor. As expected, our simple implementation of SWEET takes more time than Tor, however, it provides a sufficient performance for normal web browsing. This is in particular significant considering the strong *availability* of SWEET compared to other circumvention systems. Additionally, further optimizations on SWEET server's proxy will improve the performance. Our techniques are also amenable to standard methods to improve web latency, such as plugin-based caching and compression, which can make web browsing tolerable in high delay environments [8].

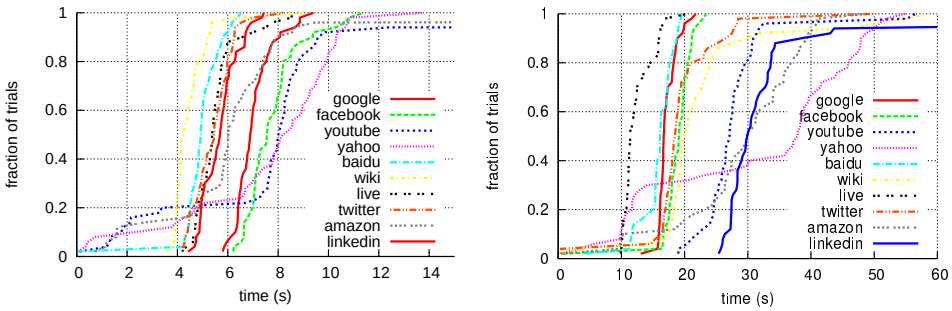

Fig. 6. The CDF of (a) the time to the first appearance (TFA) and (b) the total browsing time (TBT) using SWEET.

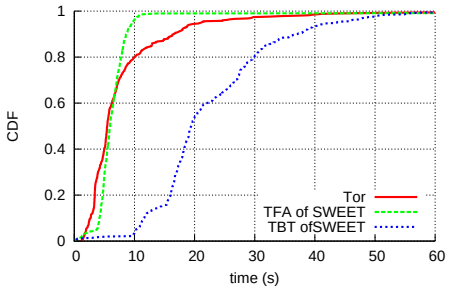

Fig. 7. Comparing the average latency of SWEET and Tor.

5.2 Traffic analysis

A powerful censor can perform traffic analysis to detect the use of SWEET, e.g., by comparing a user's email communications with that of a typical email user. As a result, a SWEET user who is concerned about unobservability needs to ensure that her SWEET email communications mimic that of a normal user. It should be mentioned that such traffic analysis is expensive for censors considering the large volume of email communications; it is estimated[7] that 294 billion emails were sent per day in 2011.

Figure 8 shows the number of emails sent and received by a SWEET client to browse different websites. We observe that for any particular website the number of emails does not change at different runs. As can be seen, most of the web sites finish in less than three SWEET emails in each direction. The exception is the Yahoo web page as it contains many web objects, hosted by different URLs (note that the number of emails affects the latency performance only *sub-linearly*, since some emails are sent and received simultaneously.). Also, the average number in each way of a connection is about 4 emails. A recent study [29] on email statistics predicts that an average user will send 35 emails and will receive 75 emails per day in 2012 (the study predicts the numbers to increase annually). In addition, membership in mailing lists[8] and Internet groups[9][10] is popular among Internet users, producing even more emails by normal email users. As an indication of the popularity of such services, Yahoo in 2010 announced[11] that 115 million unique users are collectively members of more than 10 million Yahoo Groups. Based on the mentioned statistics, we estimate that a conservative SWEET user can perform 35-70 web downloads per day, or make 10-20 interactive web connections, while ensuring unobservability of SWEET usage. Note here users who do not fear reprisal from the government might opt to have lower unobservability in order to gain a higher communication bandwidth.

6 Conclusions

In this paper, we presented SWEET, a deployable system for unobservable communication with Internet destinations. SWEET works by tunneling network traffic through widely-used public email services such as Gmail, Yahoo Mail, and Hotmail. Unlike recently-proposed schemes that require a collection of ISPs to instrument router-level modifications in support of covert communications, our approach can be deployed through a small applet running at the user's end host, and a remote email-based proxy, simplifying deployment. Through an implementation and evaluation in a wide-area deployment, we find that while SWEET incurs some additional latency in communications, these overheads are

[7] http://royal.pingdom.com/2011/01/12/internet-2010-in-numbers/
[8] http://gcc.gnu.org/lists.html
[9] http://groups.yahoo.com
[10] http://groups.google.com
[11] http://www.eweek.com/c/a/Search-Engines/Yahoo-Refreshes-Upgrades-Some-Products-775120/

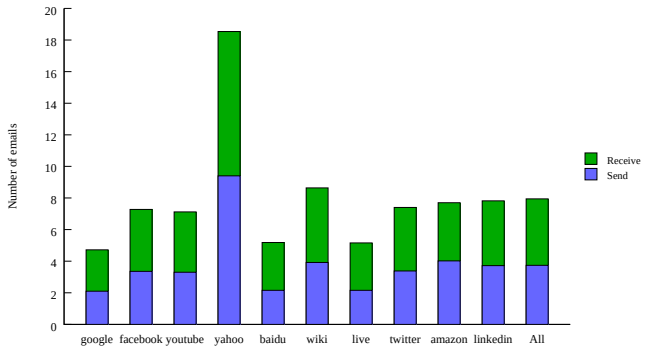

Fig. 8. The number of emails sent and received by a SWEET client to get one of the websites from Alexa's top ten ranking.

low enough to be used for interactive accesses to web services. We feel our work may serve to accelerate deployment of censorship-resistant services in the wide area, guaranteeing high availability.

References

1. The FOE project. http://code.google.com/p/foe-project/.
2. Ultrasurf. http://www.ultrareach.com.
3. Defeat Internet Censorship: Overview of Advanced Technologies and Products, Nov. 2007.
4. Defeat Internet Censorship: Overview of Advanced Technologies and Products, Feb. 2012.
5. J. Boyan. The Anonymizer: Protecting User Privacy on the Web. *Computer-Mediated Communication Magazine*, 4(9), Sept. 1997.
6. S. Burnett, N. Feamster, and S. Vempala. Chipping Away at Censorship Firewalls with User-Generated Content. In *USENIX Security Symposium*, pages 463–468. USENIX Association, 2010.
7. C. Callanan, H. Dries-Ziekenheiner, A. Escudero-Pascual, and R. Guerra. Leaping Over the Firewall: A Review of Censorship Circumvention Tools, Mar. 2010. http://www.freedomhouse.org/sites/default/files/inline_images/Censorship.pdf.
8. J. Chen, D. Hutchful, W. Thies, and L. Subramanian. Analyzing and accelerating web access in a school in peri-urban india. In *WWW (Companion Volume)*, 2011.
9. I. Clarke, T. W. Hong, S. G. Miller, O. Sandberg, and B. Wiley. Protecting Free Expression Online with Freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.
10. I. Cooper and J. Dilley. Known HTTP Proxy/Caching Problems. Internet RFC 3143, June 2001.
11. R. Dingledine and N. Mathewson. Design of a blocking-resistant anonymity system. Technical report, The Tor Project, Nov. 2006.
12. R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In M. Blaze, editor, *USENIX Security Symposium*, Berkeley, CA, USA, 2004. USENIX Association.

13. N. Feamster, M. Balazinska, G. Harfst, H. Balakrishnan, and D. Karger. Infranet: Circumventing Web Censorship and Surveillance. In D. Boneh, editor, *11th USENIX Security Symposium*, pages 247–262. USENIX Association, Aug. 2002.
14. N. Feamster, M. Balazinska, W. Wang, H. Balakrishnan, and D. Karger. Thwarting Web Censorship with Untrusted Messenger Discovery. In *International Workshop on Privacy Enhancing Technologies*, 2003.
15. A. Houmansadr, C. Brubaker, and V. Shmatikov. The Parrot is Dead: Observing unobservable network communications. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, May 2013.
16. A. Houmansadr, G. T. K. Nguyen, M. Caesar, and N. Borisov. Cirripede : Circumvention Infrastructure using Router Redirection with Plausible Deniability Categories and Subject Descriptors. In *ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, 2011.
17. A. Houmansadr, T. Riedl, N. Borisov, and A. Singer. I Want my Voice to be Heard: IP over Voice-over-IP for Unobservable Censorship Circumvention. In *Network and Distributed System Security Symposium (NDSS)*, 2013.
18. H.-C. Hsiao, T. H.-J. Kim, A. Perrig, A. Yamada, S. Nelson, M. Gruteser, and W. Ming. LAP: Lightweight anonymity and privacy. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, May 2012.
19. J. Jia and P. Smith. Psiphon: Analysis and Estimation, 2004. http://www.cdf.toronto.edu/~csc494h/reports/2004-fall/psiphon_ae.html.
20. A. Juels and J. G. Brainard. Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks. In *The Network and Distributed System Security Symposium*. The Internet Society, 1999.
21. J. Karlin, D. Ellard, A. W. Jackson, C. E. Jones, G. Lauer, D. P. Mankins, and W. T. Strayer. Decoy Routing : Toward Unblockable Internet Communication. In *Usenix FOCI*, 2011.
22. C. S. Leberknight, M. Chiang, H. V. Poor, and F. Wong. A taxonomy of Internet censorship and anti-censorship, 2010.
23. M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. RFC 1928: SOCKS Protocol Version 5, Apr. 1996.
24. M. Mahdian. Fighting Censorship with Algorithms. In P. Boldi and L. Gargano, editors, *Fun with Algorithms*, volume 6099 of *Lecture Notes in Computer Science*, pages 296–306. Springer, 2010.
25. MailMyWeb. http://www.mailmyweb.com/.
26. D. McCoy, J. A. Morales, and K. Levchenko. Proximax: A Measurement Based System for Proxies Dissemination. In G. Danezis, editor, *Financial Cryptography and Data Security*, 2011.
27. J. McLachlan and N. Hopper. On the risks of serving whenever you surf: vulnerabilities in Tor's blocking resistance design. In S. Paraboschi, editor, *8th ACM Workshop on Privacy in the Electronic Society*, pages 31–40. ACM, Nov. 2009.
28. H. M. Moghaddam, B. Li, M. Derakhshani, and I. Goldberg. Skypemorph: Protocol obfuscation for tor bridges. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.
29. S. Radicati and Q. Hoang. Email Statistics Report, 2011-2015, 2011. http://www.radicati.com/wp/wp-content/uploads/2011/05/Email-Statistics-Report-2011-2015-Executive-Summary.pdf.
30. Robert Fielding and Jim Gettys and Jeff Mogul and Henrik Frystyk and L. Masinter and Paul Leach and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999. http://www.ietf.org/rfc/rfc2616.txt.

31. Y. Sovran, A. Libonati, and J. Li. Pass it on: Social networks stymie censors. In A. Iamnitchi and S. Saroiu, editors, *7th International Conference on Peer-to-peer Systems*, Feb. 2008.
32. M. Steiner, G. Tsudik, and M. Waidner. Diffie-Hellman key distribution extended to groups. In L. Gong and J. Stern, editors, *3rd ACM Conference on Computer and Communications Security*, pages 31–37. ACM, Mar. 1996.
33. P. Syverson, G. Tsudik, M. Reed, and C. Landwehr. Towards an Analysis of Onion Routing Security. In H. Federrath, editor, *Proceedings of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*, pages 96–114. Springer-Verlag, LNCS 2009, July 2000.
34. Q. Wang, X. Gong, G. Nguyen, A. Houmansadr, and N. Borisov. CensorSpoofer: Asymmetric Communication using IP Spoofing for Censorship-Resistant Web Browsing. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.
35. Z. Weinberg, J. Wang, V. Yegneswaran, L. Briesemeister, F. W. S. Cheung, and D. Boneh. StegoTorus: A Camouflage Proxy for the Tor Anonymity System. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.
36. E. Wustrow, S. Wolchok, I. Goldberg, and J. A. Halderman. Telex: Anticensorship in the Network Infrastructure. In D. Wagner, editor, *20th Usenix Security Symposium*. USENIX Association, Aug. 2011.
37. J. Zittrain and B. Edelman. Internet Filtering in China. *IEEE Internet Computing*, 7(2):70–77, 2003.