Silent TCP Connection Closure for Cellular Networks

Feng Qian, Subhabrata Sen, and Oliver Spatscheck
AT&T Labs – Research, Bedminster, New Jersey, USA
{fengqian,sen,spatsch}@research.att.com

ABSTRACT

FIN and RST packets that close TCP connections are often delayed by timeout. In cellular networks, delayed FIN/RST packets often incur significant energy consumption overhead for handsets. On the other hand, closing TCP connection immediately after its last data transfer avoids the energy overhead, but can cause performance degradation as doing so makes reusing TCP connections difficult. To resolve such a dilemma, we propose a novel TCP extension called STC (Silent TCP connection Closure) using which both endpoints close a TCP connection silently without exchanging FIN or RST packets after timeout. Our solution is lightweight, backward-compatible, and incrementally deployable. It requires modifications to smartphone operation systems, but, if supported by cellular middleboxes, no change to remote servers. We evaluate the benefits of STC using a 10-day real trace consisting of 0.6 million LTE user sessions. When fully deployed, STC can save the overall handset radio energy consumption by up to 11.3% and reduce the network-wide signaling load by up to 6.0%.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design – Wireless communication

Keywords

TCP Connection Closure; TCP FIN; Connection Reuse; TCP Options; HTTP Keep-alive; Cellular Networks

1. INTRODUCTION

In many applications such as web browsing, it is difficult to predict when exactly data transfers of a TCP connection will finish, since a client may initiate a new request at any time after receiving the previous response. Thus a common practice is to employ an application-layer timeout to close a TCP connection. For example, HTTP keep-alive timers, which are usually statically configured, are used by almost all of today's HTTP clients and servers.

CoNEXT'13, December 9-12, 2013, Santa Barbara, California, USA.

ACM 978-1-4503-2101-3/13/12 ...$15.00.
http://dx.doi.org/10.1145/2535372.2535402.

TCP connections are usually closed by exchanging FIN packets between two endpoints. The aforementioned timeout can cause FIN packets to be delayed by seconds to minutes after the transfer of the last user data packet. In wired networks, such *delayed FIN* packets are completely inconsequential in terms of resource impact. However, in 3G/LTE networks, they may incur significant energy overhead consumed by the radio interface, as well as potentially high signaling load [11, 10], due to the interaction with the radio-layer timer that turns off the radio interface after a certain period of inactivity. Delayed FIN packets can keep the radio interface on for longer time by resetting the radio-layer timer, and even trigger an additional radio state switch by turning on the radio (§2).

The purpose of delayed closure is to increase the possibility of connection reuse, which reduces overheads such as TCP handshake, slow start, and SSL/TLS handshake especially in cellular networks with high latency. On the other hand, closing a TCP connection immediately after data transfers avoids the radio energy and signaling overhead but makes reusing the connection impossible.

This paper proposes a mechanism called STC (Silent TCP connection Closure), the first proposal that fully addresses the above dilemma. It enables mobile applications to *reuse TCP connections without incurring any resource overhead*. The high-level idea of STC is straightforward and practical: given that both endpoints usually know their own timeout for closing a TCP connection, they exchange the timeout information during the data transfer phase so both can then close the connection silently without sending any FIN packet. Often TCP RST packets can also be triggered by timeout and they can be eliminated by STC in the identical way.

In STC, each endpoint sends to its peer the timeout information embedded in a TCP option, which is always piggybacked with user data payload to ensure reliable and in-order delivery. Henceforth each side knows the timeout set by itself and the one set by its peer, and chooses the smaller one as the negotiated timeout for the TCP connection. To address the challenge of unsynchronized clocks, STC introduces an additional "protection period" after which the connection is finally torn down (§3).

STC is lightweight, backward-compatible, incrementally deployable. It exposes simple interfaces to minimize its incurred complexity at the application layer. Given that the vast majority of today's mobile apps use HTTP [13], those apps do not need to undergo any change since HTTP keep-alive is transparently handled by the underlying library, which only needs to be slightly modified to incorporate STC. In addition, by upgrading middleboxes deployed in 3G/LTE networks [10], STC can be completely transparent to servers too, thus making all changes be confined within cellular networks (§4.3).

We evaluate the benefits of STC using a 10-day trace of 0.6 million user sessions from a commercial LTE network. When fully

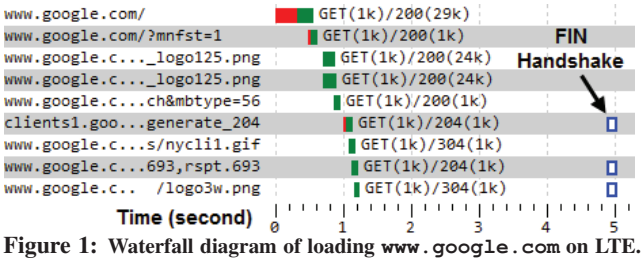

Figure 1: Waterfall diagram of loading `www.google.com` on LTE. ■=TCP Handshake ■=Object Transfer □=FIN handshake.

deployed, STC can reduce the *network-wide* handset radio energy consumption by up to 11.3% and the overall signaling load by up to 6.0% (§5). STC can be applied to other access technologies using resource control mechanisms similar to those in cellular networks.

2. MOTIVATION

Our proposal is motivated by three facts described below.

■ **Observation 1: Delayed FIN/RST consume radio energy and incur signaling load in cellular networks.** It is well known that in 3G and LTE networks, there exists a radio-layer timeout for turning off the radio interface after a period of network inactivity [11, 9]. Such a timeout, called *tail time*, prevents frequent radio state switches (*i.e.,* the signaling overhead). The tail timer is reset to a fixed value by any incoming or outgoing packet. When the timer ticks down to zero, the radio interface is turned off. Note that the cellular radio power contributes to as high as 1/3 to 1/2 of the total handset power usage for 3G and LTE [11].

A delayed FIN or RST lengthens the radio-on time by resetting the tail timer. Figure 1 shows a real example of loading `www.google.com` on a Samsung Galaxy S III smartphone over LTE. The waterfall diagram indicates that the three TCP connections involved are closed at 5.0 sec, while the page loading finishes at 1.2 sec. Given that the tail timer is measured to be 11.0 sec, the overall radio-on time is 5.0+11.0=16.0 sec. If all connections are closed right after data transfers, the radio-on time will be reduced by 22.5% – only 1.2+0.2+11.0=12.4 sec (closing the connection takes 0.2 sec). But doing so prevents TCP connections from being reused for future transfers, leading to performance degradation especially in cellular networks with high latency.

If a FIN/RST delay is longer than the tail time and no other concurrent data transfer exists, the delayed FIN/RST will trigger an additional radio state switch by turning on the radio. In 3G/LTE, turning on the radio incurs up to 30 messages over control channel, causing signaling overhead [5].

■ **Observation 2: Delayed FINs are usually controlled by application-layer keep-alive timers.** Almost all web servers have configurable keep-alive timeout settings *i.e.,* the fixed amount of time the server will wait for for a subsequent request before closing the connection. For example, Apache 2.4 uses 5 seconds and Microsoft IIS 7 uses 120 seconds by default. Many servers also explicitly inform clients about their timeout using the `Keep-Alive:timeout` response header. The client should attempt to retain a connection for at least as long as indicated [3].

Mobile clients also use timers to close TCP connections. For example, the popular `org.apache.http.client` library allows developers to explicitly set the keep-alive timer for each connection. We also examined the implementation of the idle TCP connection cache (`sun.net.www.http.KeepAliveCache`) used by Java HTTP libraries. By default, each (remote host, remote port) pair can have at most five idle connections. Their timeout value is determined by the `Keep-Alive:timeout` response header, or a statically configured parameter (5 seconds by default) if the header

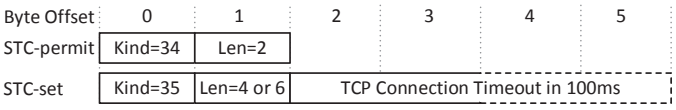

Figure 2: The TCP options introduced by STC.

is not provided. When the client needs to fetch a new URL, it tries to reuse an unexpired connection in the cache. For every 5 seconds, a daemon thread scans the whole cache and closes idle connections that are timed-out. The above implementations are extensively used by Android applications.

Note the above application-layer keep-alive should not be confused with TCP keep-alive, which probes whether a TCP connection is still alive after a long period of idle time (*e.g.,* 1 hour) if neither side closes it.

■ **Observation 3: Delayed FIN/RST are widespread.** Based on studying a 10-day LTE trace consisting of 0.6 million user sessions, we found 51% of all TCP connections (72% of connections carrying HTTP) have FIN/RST delays of at least 1 second. We detail the measurement in §5.

3. SILENT TCP CONNECTION CLOSURE

The current FIN-based scheme for closing TCP connections poses a dilemma: using delayed FINs often wastes radio energy and incurs signaling load, while prematurely tearing down connections makes reusing them impossible. STC addresses this by enabling mobile applications to reuse TCP connections without incurring any resource overhead. Specifically, given that both a mobile app and the server usually know when they will close a TCP connection (§2), in STC, both sides exchange connection timeout information during the data transfer phase so both can then close the connection silently without sending any FIN packet. Often TCP RST packets can also be triggered by timeout and they can be eliminated by STC in the same way. We next describe how key challenges are addressed by STC in detail.

■ **Challenge 1: Reliable and in-order delivery of the timeout information.** To achieve this, the timeout information is encapsulated into TCP options that are always piggybacked with TCP *user data* (or SYN/SYN-ACK), thus providing automatic guarantee of reliability and ordering.

As shown in Figure 2, STC introduces two TCP options. The `STC-permit` option is used to negotiate during a TCP handshake whether STC will be used. STC is only used if supported by both sides *i.e.,* the client sends `STC-permit` in SYN and the server includes it in SYN-ACK, making STC incrementally deployable.

The `STC-set` option can be sent within an STC-enabled TCP connection by either side. It contains a value x, which notifies the receiver that the sender has chosen a connection timeout of $x * 100$ milliseconds. x usually takes two bytes, supporting a timeout period of up to 6553.5 seconds, beyond which a four-byte value can be used. Note that most TCP packets do not carry `STC-set`, which is only used when the timeout is changed or set for the first time.

As mentioned before, `STC-set` messages must be piggybacked with data packets (or SYN/SYN-ACK) so that they can be retransmitted when lost, and be delivered in the same order as they are sent. Doing so also guarantees that `STC-set` does not reset the radio-layer tail timer by itself (§2) since `STC-set` never causes any additional packet transfer, making STC not incur any additional resource overhead.

STC disallows updating the connection timeout without sending user data. Supporting such an uncommon use case requires more complex changes to the TCP protocol *e.g.,* `STC-set` messages may need sequence numbers to ensure their delivery order. To update

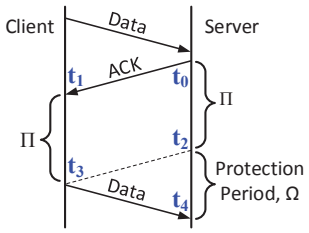

Figure 3: An example illustrating the protection period.

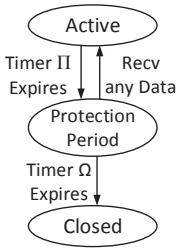

Figure 4: STC state transitions.

the timeout, an app can wait until the next data transfer, or establish a new connection.

■ **Challenge 2: Lightweight protocol.** For the ease of presentation, we focus on the client-side logic. The server-side logic is identical as STC does not distinguish between a client and a server.

For an STC-enabled TCP connection, the client's TCP stack maintains two variables T_{client} and T_{server} corresponding to the connection timeout specified by the client (itself) and the server, respectively, using **`STC-set`**. The client's TCP also maintains an inactivity timer Π, which is reset by any network activity of the connection. When Π ticks down to zero, the connection is about to be closed silently. The detailed logic is as follows.

1. Initially, both T_{client} and T_{server} are set to ∞. Π is also set to ∞ (*i.e.,* never expires).

2. When the client successfully receives an **`STC-set`** with timeout x, it sets T_{server} to x. Here "successfully receive" means the expected sequence number matches the sequence number of the associated data packet, which is then delivered to the upper layer. Similarly, when the client successfully sends an **`STC-set`** with timeout x, it sets T_{client} to x. Here "successfully send" implies that the client receives an ACK packet that acknowledges the reception of the data packet associated with **`STC-set`**. The above mechanisms ensure that an endpoint applies its peer's timers in the same order in which its peer sets them, since their associated data packets are delivered in order.

3. Usually both SYN and SYN-ACK contain **`STC-set`**. So at the client side, both T_{client} and T_{server} are initialized when SYN-ACK is received. At the server side, T_{client} (maintained by the server) is initialized when SYN is received, and T_{server} is initialized when the last ACK in the TCP three-way handshake is received.

4. The inactivity timer Π is reset to $\min\{T_{\text{client}}, T_{\text{server}}\}$ when the connection sends or receives *any* packet associated with it. Resetting Π indicates the network activity of the connection. Step 4 always happens after Step 2 and 3 if the packet contains an **`STC-set`** message.

5. When Π expires, the TCP connection is closed silently after a short duration (described next).

■ **Challenge 3: Unsynchronized timers.** Due to the network latency and its variability, the inactivity timer Π at both sides are not strictly synchronized, leading to potential inconsistency that Π at one side has expired but Π at the other side has not. In order to prevent an undesired situation where an endpoint sends data to its remote peer who has already closed the connection silently (we call this *"undesired silent closure"*), on either side, when Π expires, the connection is not closed immediately. Instead, it will enter a *protection period* during which the connection can only *receive* incoming data but not send any data. However, upon the reception of any data, the connection immediately exits the protection period and becomes fully active by resetting the Π timer to $\min\{T_{\text{client}}, T_{\text{server}}\}$. The protection period has small impact on applications, as will be described shortly.

The connection will finally be closed when the protection period ends with no incoming packet. The duration of the protection period is controlled by another inactivity timer Ω. To determine how to set this timer, consider an example shown in Figure 3. The client and server reset their Π timers at t_1 and t_0 respectively. Right before the client's Π timer expires at t_3, the client sends a data packet, which arrives the server at t_4. However, the server's Π timer has already expired at t_2. Therefore, in order to eliminate the possibility of receiving data after silently closing a connection (*i.e.,* an undesired silent closure), the server's protection period should be at least $t_4 - t_2 = (t_4 - t_3) + (t_1 - t_0) = \text{RTT}$. In practice, the Ω timer can be statically or dynamically set to be far greater than a normal RTT. Figure 4 summarizes the state transitions in STC.

When experiencing packet losses, the Π timer difference between two endpoints can be larger. We therefore add extra rules to prevent an undesired silent closure due to poor network condition. In Figure 4, even if the Π timer expires, the transition from an active connection to the protection period will not happen if any of the following conditions holds: *(i)* there are unacknowledged packets, *(ii)* there are unreceived packets (*i.e.,* receiving packet(s) with higher sequence number(s) than expected), *(iii)* either the TCP sending buffer or the receiving buffer is not empty. This can happen when the end-host's CPU is busy.

The protection period ensures that STC does not cause data packet loss at the end of a TCP connection due to a lack of explicit FIN. However, if the protection period is not long enough, an undesired silent closure can happen. In that case, the endpoint that observes the incoming data will reply with a TCP RST without acknowledging the data, so the remote peer's TCP stack will close the corresponding connection and notify the application about the failed delivery, and the application will establish a new connection so the correctness of the application logic is not affected. However, we expect this will happen extremely rarely given the aforementioned measures against an undesired silent closure.

4. DISCUSSIONS

We discuss implementation and deployment issues of STC.

4.1 Minimize Changes to Applications

STC provides two interfaces to applications:

- Socket options for enabling STC and setting connection timeout to be piggybacked with the next user data.
- Interface for applications to query the state of a connection (Figure 4) and its remaining lifetime (*i.e.,* the Π timer).

The above interfaces can be achieved by **`setsockopt()`** and **`getsockopt()`** system calls. Other socket APIs such as **`send()`** and **`recv()`** are not changed. Also, traditional FIN and RST can still be used at any time (*e.g.,* by explicitly calling **`close()`**), and they override the STC mechanism. Using delayed FIN is discouraged, but RST is still useful in case of an error.

We next provide examples to illustrate how existing applications can be easily adapted to STC when it is enabled. *(i)* **A web server** using STC does not explicitly close a connection by calling **`close()`**. Instead, when a connection is silently closed by TCP, the server performs clean-up as if the connection is explicitly closed. A connection in the protection period is treated the same as an active connection since a web server does not initiate an HTTP transaction so it naturally only receives data during the protection period. *(ii)* **A client HTTP library** will be changed in a similar way except that a connection in the protection period should not be used to send a request. *(iii)* **Most smartphone apps** use HTTP [13].

They will remain untouched if the underlying libraries manage TCP connections transparently. Almost all popular HTTP libraries satisfy such a requirement.

4.2 The TIME_WAIT State

In TCP, an *active closer* who initiates the connection close will eventually enter a state called `TIME_WAIT` before closing the connection. In `TIME_WAIT`, the endpoint waits for a fixed amount of time (twice the maximum lifetime of a packet) during which the four-tuple (src IP/port, dst IP/port) defining the connection cannot be reused. This eliminates an undesired situation where a packet belonging to the old connection is delivered later to a newly created connection with the same four-tuple. There is no need for the other endpoint to stay in `TIME_WAIT`. If both sides send FIN simultaneously, then both will hold `TIME_WAIT`. In STC, since it is impossible to tell which side is the active closer, a new rule can be applied: in a silent closure, the endpoint that sends SYN will hold `TIME_WAIT` while the other endpoint will directly move to the final `CLOSED` state. If both send SYN simultaneously, then both will hold `TIME_WAIT`.

4.3 Interaction with Middleboxes

A legacy NAT box not recognizing STC does not affect the functionality of STC. But when silently closed, a connection will not be removed from the connection table of the NAT. Instead, it will be eventually removed by timeout at the NAT, ranging from less than 5 mins to more than 30 mins in today's cellular networks [12]. This is largely not an issue since NAT scales well with the number of entries. It uses efficient hash-based lookup [7], and for a commodity Cisco NAT box, 10K entries consume only 3 MB of RAM [2]. Note even if a connection is closed by FIN, its NAT entry will also be removed by a timeout (*e.g.,* 1 minute [2]).

Next, we show that if NAT boxes are aware of STC, additional benefits can be achieved even for *long-lived* connections. Carriers often configure their NAT boxes to have short TCP connection timeout. This affects long-lived connections used by, for example, chat apps and push notifications, which must send application-level keep-alive messages to prevent the connections from being closed by NAT timeout. Such keep-alive messages, if frequently sent, can cause significant handset energy waste due to the tail time [12]. NAT boxes use timeout because they do not know when connections will close. STC instead provides a way to explicitly inform NAT of the lifetime of a connection, thus making it unnecessary for applications to send frequent keep-alive messages.

Proxies. Cellular carriers also deploy general-purpose proxies for purposes such as caching and compression [10]. From the perspective of TCP connection management, they could be classified into two types: *split proxies* (SP) and *non-split proxies* (NSP). The former splits an end-to-end TCP connection into two, one between the handset and the proxy and the other between the proxy and the remote server. An SP makes the split transparent to handsets by spoofing its IP address to be the server's IP address. In contrast, an NSP is less intrusive. It does not split TCP connections traversing it but can modify packets on the fly.

Carriers have strong incentives to make their proxies STC-capable: regardless of its type, as long as a proxy supports STC, there is no need to modify remote servers (or any network element on the upstream path of the proxy). Therefore *all changes can be confined within cellular networks*. But a server can still choose to support STC for customized control over the connection timeout.

Specifically, when a split-proxy (SP) is present, it must recognize STC otherwise only traditional FIN/RST can be used even if both the handset and the server support STC. Adding STC support to an SP is the same as upgrading a server, as described before. For a non-split-proxy (NSP), STC can function if both endpoints support it but the NSP does not (similar to the case of NAT). Making an NSP STC-capable is also easy. Assume a handset sends a SYN with `STC-permit` and usually `STC-set`. If the original server sends a SYN-ACK with `STC-permit`, the NSP can then let the server handle STC. Otherwise, since the server does not support STC, the NSP modifies the original SYN-ACK from the server by adding `STC-permit` and `STC-set`. Also, in the data transfer phase, the NSP attaches `STC-set` to data packets from the server based on its own policy (setting T_{server} to be longer than T_{client} is recommended). When the connection is closed silently, the NSP sends a RST to the STC-unaware server on behalf of the handset but there is no packet exchange between the NSP and the handset[1].

4.4 Security

To our knowledge, STC does not bring in new security issues. Also, firewalls can be easily configured to disable STC by removing the `STC-permit` TCP option (*e.g.,* just one line of command on Cisco firewalls [1]). This makes the use of STC fully controllable.

4.5 Alternative Solutions

STC reduces the resource overhead of delayed FIN/RST by eliminating them at TCP layer using a lightweight protocol. The overhead can also be potentially reduced at other layers. We next discuss a few alternative approaches and their limitations.

Reducing the tail timer or the HTTP keep-alive timer mitigates the resource overhead issue. In particular, setting the keep-alive timer to be smaller than the radio-layer tail timer helps avoid the signaling overhead caused by delayed FIN/RST. However, there are side effects. Reducing the tail timer leads to higher signaling load and worse user experiences [9], and having a smaller HTTP keep-alive timer can make reusing TCP connections more difficult.

Using fewer connections by multiplexing multiple HTTP transactions into one connection can mitigate the delayed FIN issue. A well-known realization of this approach is the Google SPDY protocol that opens one connection per domain [8]. However, it is quite common that a website contains multiple domains, thus delayed FIN handshakes in SPDY can still cause resource overhead. Regarding to performance, a recent study [8] reveals that compared to HTTP, SPDY is more likely to incur head-of-line blocking due to its unexpected interaction with TCP.

Piggyback FIN with delay-sensitive transfers. At the application layer, various traffic shaping and scheduling techniques have been proposed to make cellular data transfers more resource-efficient. For example, delay-tolerant transfers can be delayed [6], and predictable transfers (*e.g.,* checking emails) can be prefetched so that those transfers can be piggybacked with delay-sensitive transfers or be batched in fewer data bursts, thus reducing the radio-on time. Delayed FINs are to some extent delay-tolerant, and their piggyback or batching can be achieved at the application layer by calling `close()` intelligently. However, piggybacking FIN with other transfers or batching multiple FIN handshakes together is not always possible, and doing so can increase the complexity of the application logic.

Fast Dormancy is a feature included in 3GPP since Release 7 [4]. It allows a handset to send a control message to the RAN to immediately turn the radio state to idle without experiencing the tail time. The handset can therefore invoke fast dormancy right after a

[1]If a remote server sends data to the proxy (either SP or NSP) that is in the protection period, the proxy drops the data and sends RST to the server. This does not happen for a web server and is expected to be very rare for other types of servers if a long timeout is used.

Table 1: TCP connection termination status (across all connections).

Method	Initiator	Proxy Flows	Non-proxy Flows
FIN	Handset	77.0%	59.0%
	Server/Proxy	15.0%	19.4%
RST	Handset	5.6%	2.8%
	Server/Proxy	0.4%	2.6%
Not closed within 1 hour		2.1%	16.2%

data transfer to save radio energy. However, a key limitation of this approach is that the incurred signaling load caused by frequently switching on the radio may be too high to be handled by the radio access network [5].

5. TRACE-DRIVEN EVALUATION

We evaluate the benefits of STC using real LTE traces, which are large packet header data collected from a commercial LTE carrier. The data covers a fixed set of 22 eNBs at a large metropolitan area in the U.S. The data collection was between Oct 12 and Oct 21, 2012. For each packet, we recorded its IP and transport-layer headers and a 64-bit timestamp. During the 10 days, we obtained 3.8 billion packets, corresponding to 2.9 TB of LTE traffic.

The data collection point is between the LTE radio access network (RAN) and the core network gateway (SGW/PGW). TCP traffic from or to server port 80 or 8080 will traverse a *split proxy* (§4.3). Therefore, the network path of an 80/8080 packet is: handset↔RAN↔data collector↔SGW/PGW↔Split Proxy↔NAT/Firewall↔Internet. The path for all other packets is the same except that the proxy is skipped.

To protect the privacy and anonymity of network users, we did not collect or use any personally identifiable information. Instead, we used hashed private IP addresses as approximated subscriber IDs since the mappings from subscribers to private IPs of the studied carrier are very stable (a handset changes its private IP at the interval of several hours). We therefore separate the trace into *user sessions* each consisting of packets sent to and received from a particular private IP address. We use an idle period of 1 hour to determine the end of a user session. Changing this threshold to 30 minutes or 2 hours has very small impact on the results to be described. Within a user session, individual TCP flows are identified based on the four-tuple of src/dst IPs and src/dst port numbers. Overall we obtained 0.59 million user sessions and 51 million TCP flows during the 10-day data collection period. TCP contributes to 97% (95%) of the total bytes (flows). Within TCP, 77% (50%) of its bytes (flows) traverse the proxy.

5.1 Characterizing TCP Connection Closure

Table 1 shows how TCP connections (flows) in the dataset are closed. Recall that the two endpoints of a non-proxy flow are a handset and the remote server, while a captured proxy flow with server port 80/8080 is between a handset and the split proxy whose HTTP keep-live timer overrides the one of the original web server. Table 1 indicates that connections are usually terminated by FINs (92.0% of proxy flows and 78.4% of non-proxy flows). Regarding to the initiator, most connections (82.6% of proxy flows and 61.8% of non-proxy flows) are closed by the handset. About 16.2% of non-proxy flows are not closed after being idle for at least one hour, the threshold for separating user sessions. Such flows mostly consist of HTTPS (port 443, 65.9%), IMAP (port 993, 10.7%), and XMPP messaging (port 5223, 7.4%).

Figure 5(a) and (b) plot the distributions of closure delays across non-proxy and proxy flows, respectively. The closure delay is defined to be the timestamp difference between the last data packet

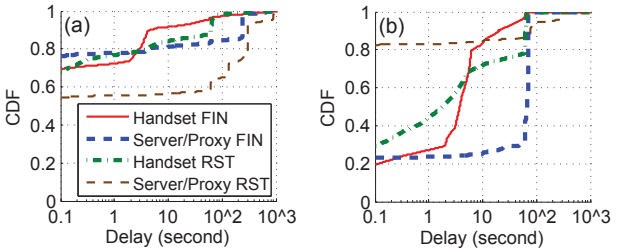

Figure 5: (a) FIN/RST delays across *non-proxy* flows (b) FIN/RST delays across *proxy* flows. Both subfigures have the same legends.

and the first FIN/RST packet in a flow[2]. We have the following observations. *(i)* Within proxy flows, 72.8% (76.5%) of handset-FIN-terminated (proxy-FIN-terminated) flows have delays longer than 1 second. For non-proxy flows, the corresponding percentages are only 27.7% for handset-FIN-terminated flows and 22.0% for server-FIN terminated flows. Given that non-proxy flows are dominated by HTTPS (port 443, 82.7%) and IMAP (port 993, 7.0%), this indicates that unlikely HTTP clients/servers that usually employ delayed FINs, HTTPS and IMAP clients/servers tend to close TCP connections immediately after data transfers. *(ii)* In Figure 5(b), the value of about 1 minute dominates the connection timeout maintained by the split proxy, while for server-initiated FINs in Figure 5(a), we observe a cluster of 4-minute timeout. In addition to the radio energy waste, such long FIN-delays also incur signaling overhead as they are much longer than the radio-layer tail time. *(iii)* TCP RST is also often delayed. In particular, within proxy flows, 56% of handset-issued RST packets are delayed by at least 1 second. We also observe clusters of RST delays of 1, 2, and 5 minutes in both figures, implying that a large fraction of RST packets are also triggered by timeout instead of unexpected errors. Overall, delayed FIN/RST are *widespread*. 51% of all TCP flows (72% of proxy flows carrying HTTP) have FIN/RST delays of at least 1 second.

The Protection Period should be set to be at least one RTT (Figure 3). The 99.8-percentile of non-proxy flow RTTs and the 99.9-percentile of proxy flow RTTs are measured to be 4.0 sec and 3.8 sec, respectively. Therefore setting the protection period duration (*i.e.,* the Ω timer) to be 5 sec is robust enough to prevent from receiving data after closing the connection. An end-host can also dynamically set the Ω timer based on its RTT estimation.

5.2 Quantifying Benefits of STC

Next, we quantitatively study the network-wide benefits of STC by comparing the resource consumption of the original trace and the modified trace with FIN packets removed, as if STC is used between handsets and proxy/servers. The analysis is performed as follows. *(i)* For each user session u, we feed it into an LTE power model, which computes the radio energy consumption $E(u)$, the radio-on time $R(u)$, and the signaling load $S(u)$ by simulating the LTE Radio Resource Control (RRC) state machine [9]. $S(u)$ is defined as the number of radio state transitions from the idle to the connected state. The handset parameters are configured for a state-of-the-art LTE smartphone (HTC Thunderbolt), and the radio access network parameters correspond to those used by the studied carrier, whose tail timer is measured to be 11 seconds. Applying a 3G UMTS/HSPA power model [11] yields slightly more savings. *(ii)* We remove all FIN packets from u and get the resultant trace u'. The power model then takes u' as input and computes $E(u')$, $R(u')$, and $S(u')$. *(iii)* The radio energy

[2]For flows without user data, the closure delay is the timestamp difference between the first FIN/RST and its previous packet.

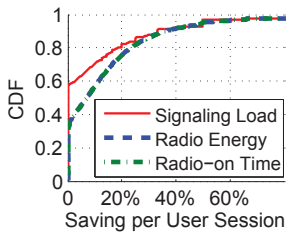

Figure 6: Savings across user sessions.

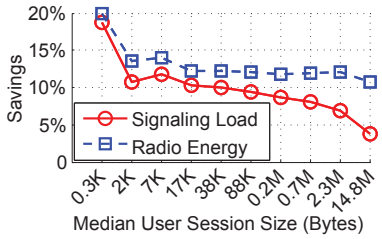

Figure 7: Savings vs. user session size.

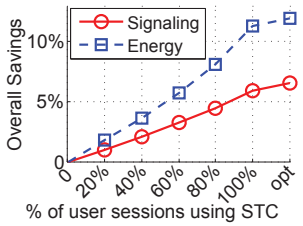

Figure 8: Incremental deployment.

saving is quantified as $(E(u) - E(u'))/E(u)$. The savings of the radio-on time and the signaling load are calculated in similar ways. We conservatively do not remove any RST packets although many of them, which are also triggered by timeout, can be eliminated by STC in practice. Also, a user session often includes multiple concurrent TCP connections within which one may keep the radio on, thus reducing the benefits of STC applied to other connections. This scenario is captured based on the real trace in our evaluation.

Savings per User Session. Figure 6 plots the distributions of savings across all user sessions. For the radio energy and the radio-on time, which follow very similar distributions, the savings range from 0 to 80%. On one hand, about 38% of user sessions have little radio energy savings (less than 1%), either because they have few TCP connections with delayed FINs, or because delayed FINs are overlapped with other concurrent TCP/UDP data transfers. In the latter case, removing the delayed FINs brings no reduction in any of the three metrics. On the other hand, for some user sessions, the savings are non-trivial or even very significant. The 50, 60, 70, 80, and 90 percentiles of radio energy (or radio-on time) savings are 6%, 10%, 15%, 22%, and 34%, respectively. The reduction of the signaling overhead follows a similar distribution but the overall saving is less.

Savings vs. User Session Size. Figure 7 studies the correlation between savings brought by STC and the user session size. We separate all 0.59 million user sessions sorted by their total bytes into 10 groups each having 59K sessions. We then compute the savings for *all* sessions in each group whose median session size is shown on the X axis in Figure 7. Statistically, small user sessions tend to have higher savings, especially for the signaling overhead. For large sessions that are more likely to have long-lived TCP connections, their long flow duration often diminishes the savings brought by silent connection closure.

Incremental Deployment. Figure 8 examines the benefits of deploying STC incrementally by eliminating FINs for only $x\% \in \{0, 20\%, ..., 100\%\}$ of randomly chosen user sessions. Figure 8 indicates the savings increase linearly as x increases. When fully deployed, the *network-wide* handset radio energy consumption and signaling load can be reduced by 11.3% and 6.0%, respectively. For most non-heavy users not belonging to the long tail of the user session size distribution, their average savings will be higher as indicated in Figure 7. In Figure 8, "`opt`" on the X axis corresponds to a scenario where in addition to FINs, all RSTs are also removed. In that case, the overall radio energy and signaling load savings increase to 11.9% and 6.5%, respectively.

Performance Improvement. For TCP flows carrying HTTPS traffic, more than 70% of them are observed to be closed prematurely right after the last data transfer. STC can let those connections be more efficiently reused without incurring any resource overhead. Connection reuse benefits in particular SSL/TLS over TCP where the high SSL/TLS handshake overhead can be significantly reduced. We will quantify the performance benefits of STC in future work.

6. RELATED WORK AND CONCLUSION

The adverse impact of delayed FIN/RST on cellular resource utilization was first pinpointed by the ARO tool on individual mobile apps [11]. A recent study [10] measures the timing gap between the last data packet and the last packet of TCP flows, using the same dataset studied in this paper. Both works qualitatively or quantitatively indicate the importance of the delayed FIN/RST problem, but neither proposed a concrete and effective solution as we did in this paper. In §4.5, we discussed a few alternative approaches that can potentially reduce the resource overhead of delayed FIN/RST, such as using fewer connections [8], piggyback/batching [6], and fast dormancy [4, 5]. All of them exhibit some limitations that are avoided in our STC design.

To conclude, STC is the first proposal fully addressing the delayed FIN/RST problem in cellular networks. It is lightweight, backward-compatible, and incrementally deployable. It enables mobile apps to reuse TCP connections without incurring any resource overhead while prior to STC it was difficult to achieve both advantages at the same time. We are currently working on a prototype implementation of STC.

7. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and especially Marco Mellia for shepherding the paper.

8. REFERENCES

[1] Cisco ASA 5500 Series Configuration Guide. `http://www.cisco.com/en/US/docs/security/asa/asa84/configuration/guide/conns_connlimits.html`.

[2] Cisco NAT FAQ. `http://www.cisco.com/en/US/tech/tk648/tk361/technologies_q_and_a_item09186a00800e523b.shtml`.

[3] HTTP Keep-Alive Header. `http://tools.ietf.org/id/draft-thomson-hybi-http-timeout-01.html`.

[4] UE "Fast Dormancy" behavior. 3GPP discussion and decision notes R2-075251, 2007.

[5] P. K. Athivarapu, R. Bhagwan, S. Guha, V. Navda, R. Ramjee, D. Arora, V. N. Padmanabhan, and G. Varghese. RadioJockey: Mining Program Execution to Optimize Cellular Radio Usage. In *Mobicom*, 2012.

[6] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications. In *IMC*, 2009.

[7] S. S. Chugh. Impact of Network Address Translation on Router Performance. Master's thesis, Virginia Polytechnic Institute and State University, 2003.

[8] J. Erman, V. Gopalakrishnan, R. Jana, and K. Ramakrishnan. Towards a SPDY'ier Mobile Web. In *CoNEXT*, 2013.

[9] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *Mobisys*, 2012.

[10] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck. An In-depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance. In *SIGCOMM*, 2013.

[11] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Profiling Resource Usage for Mobile Applications: a Cross-layer Approach. In *Mobisys*, 2011.

[12] Z. Wang, Z. Qian, Q. Xu, Z. M. Mao, and M. Zhang. An Untold Story of Middleboxes in Cellular Networks. In *SIGCOMM*, 2011.

[13] Q. Xu, J. Erman, A. Gerber, Z. M. Mao, J. Pang, and S. Venkataraman. Identifying Diverse Usage Behaviors of Smartphone Apps. In *IMC*, 2011.