A Disruption-tolerant Transmission Protocol for Practical Mobile Data Offloading

Younghwan Go, YoungGyoun Moon, Giyoung Nam, and KyoungSoo Park

Department of Electrical Engineering, KAIST
Daejeon, South Korea
{yhwan, ygmoon, giyoung}@ndsl.kaist.edu, kyoungsoo@ee.kaist.ac.kr

ABSTRACT

The explosive popularity of smartphones and mobile devices drives massive growth in the wide-area mobile data communication. Unfortunately, the current or near-future 3G/4G networks are deemed insufficient to meet the increasing data transfer demand. While opportunistic offloading of mobile data through Wi-Fi is an attractive option, the existing transport layer would experience frequent disconnections due to mobility, making it hard to support seamlessly reliable data delivery. As a result, many mobile applications either depend on ad-hoc downloading resumption mechanisms or redundantly re-transfer the same content when disruptions happen.

In this paper, we present DTP, a disruption-tolerant, reliable transport layer protocol that masks the failures of the preferred network. Unlike previous disruption/delay-tolerant protocols, DTP provides the same semantics as TCP on an IP packet level when the mobile device is connected to a network while providing the illusion of continued connection even if the underlying physical network becomes unavailable. This would help the mobile application developers to focus on the application core rather than addressing the frequent network disruptions. It would also greatly reduce the phone network costs both to ISPs and end users. Our current implementation in UDP shows a comparable performance to that of TCP in network, and it greatly reduces the delay and power consumption when the mobile devices frequently switch from one network to another.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Store and forward networks; C.2.1 [**Network Architecture and Design**]: Wireless communication; C.2.2 [**Network Protocols**]: Applications

General Terms

Design, Performance

Keywords

Delay Tolerant Network, Wi-Fi Offloading, Mobility

MobiOpp'12, March 15–16, 2012, Zürich, Switzerland

1. INTRODUCTION

Recent advancement in cell-phone networks and smartphones has brought massive growth in the mobile data communication. The number of mobile network users is expected to surpass that of the wired Internet within the next four years [1] and the global traffic volume is predicted to consume 6.3 Exabytes per month in 2015, a 26-fold increase from that of 2010 [2]. However, the existing 3G or Long Term Evolution (LTE) networks in the near future are unlikely to provide as much bandwidth as in the wired Internet, and the capacity shortage is becoming a serious barrier to advancing the mobile data communication.

There have been a number of works that address the capacity overloading problem. One end of the efforts is to increase the physical capacity by reducing the cell size or by intelligent multiplexing of the shared radio medium [3,4]. However, these approaches have fundamental limitations when the aggregate network demands exceed the physical capacity. The other end of the line focuses on adopting the hybrid usage of 3G and much higher-bandwidth networks such as the wired Internet through Wi-Fi. The idea is to offload the 3G mobile data transfer via Wi-Fi opportunistically while using the 3G networks as a backup medium to meet the transfer deadline [5–8]. This opportunistic Wi-Fi offloading is an attractive option especially in urban areas with high Wi-Fi availability, with the potential of reducing the 3G data bandwidth consumption via delay-tolerant networking (DTN) [9]. We believe that many non-interactive data-intensive applications such as podcast [10, 11], TV episode or movie downloading [12], or personal storage synchronization [13] could benefit from it.

In this paper, we promote delay-tolerant, opportunistic Wi-Fi offloading of 3G mobile data from a practical point of view. Our goal is to support Wi-Fi offloading with little or no change to the current applications or underlying networks. We observe that the 3G or Wi-Fi networks show stable behavior most time while the mobile devices are connected, but one needs to handle network disruptions preferably in a transparent manner when the mobile devices switch from one network to another. According to recent measurements, 87% of the entire smartphone usage occurs while the users are on the move, implying frequent switches between multiple networks [14].

One approach is to have the applications handle network disruptions by themselves. In fact, some applications already support download resumption when they change their network attachments. However, this necessitates an ad-hoc implementation of download resumption in each application (e.g., HTTP byte-range queries, CGI parameter passing, and so on), which cannot be easily reused by other applications. For dynamically-generated contents, applications may not be able to determine where to resume down-

loading or end up with re-downloading the whole content on a new connection if the IP address of the device changes. Another approach, which we favor in this work, is to transparently handle the network disruptions in the transport layer. Since the majority of mobile applications use TCP, if we make TCP disruption-tolerant, many non-interactive applications could benefit from transparent Wi-Fi offloading with minimal change. This would also ease the burden on the application developers so that they focus on the core program logic rather than handling network failures or disruptions due to device mobility.

We present the design and implementation of DTP, a disruption-tolerant transport layer protocol that transparently masks network failures from the application layer. On a high level, DTP works similarly to TCP when the mobile device is attached to a network but it provides the illusion of continued connection to the applications even when the underlying network is unavailable. This way, DTP allows the mobile applications to exploit Wi-Fi offloading without requiring them being DTN-aware. Unlike previous DTN protocols [15–18], DTP supports reliable data delivery on a packet level and it does not require any special support from the network infrastructure.

The key technical challenge in DTP is how we manage the connection when the physical network switches between on and off. Instead of binding the connection on the four connection tuples (source and destination IP addresses and port numbers), DTP binds the connection to a *flow ID* that is agreed at the initial connection setup time and does not change during the connection lifetime. When a mobile host moves to another network, it can resume the connection with a new IP address and a port number by cryptographically attesting that it owns the flow ID of the connection. The DTP connection closes either when both parties explicitly tear it down or when the *keep-alive duration* of the connection expires. The keep-alive duration is the estimated connection lifetime set at the connection setup time that can be updated during the course of the connection.

While DTP hides the network disruptions transparently from the application layer, it presents a few security problems. Malicious hosts may attempt to hijack a connection by resuming an interrupted one or create lots of fake states on the server side. To prevent connection hijacking, DTP exchanges a secret key at connection setup and authenticates the other end by a simple challenge-and-response protocol before resuming. To mitigate the state explosion attacks, DTP keeps a minimal state per flow (less than 200 bytes per flow), reducing the memory burden on the server.

We build the prototype of DTP as a UDP-based API library where each function has a one-to-one correspondence to a TCP socket function. Our initial evaluation shows that its performance is comparable to that of TCP on wired or Wi-Fi networks while it shows 47% and 123% better performance for moderate and large file sizes in a typical delay-tolerant setting.

2. BACKGROUND

Many smartphones and tablet PCs these days have both 3G and and Wi-Fi interfaces. The availability of 3G is typically more ubiquitous than that of Wi-Fi [5], but it is more expensive due to smaller capacity. While the next generation cell-phone networks such as LTE are being deployed, they are unlikely to catch up with the fast-growing mobile data demand in the future.

To mitigate the 3G capacity overloading problem, many data-intensive mobile applications configure Wi-Fi as the preferred interface by default, and explicitly ask for the permission from the user when it needs to switch to 3G. Recent studies show that one can benefit further from Wi-Fi offloading if we allow some delay for data transfer [5, 6]. We can offload 10% to 30% of 3G data to Wi-Fi if we disallow any interruptions in data transfer, but the offloading ratio can go up to 75% if we allow 30 minutes of delay and to 88% for 6 hours of delay [6]. While delays in real-time contents would lead to poor user experience, we find that many non-interactive applications (e.g., large-file downloading) could benefit from delayed Wi-Fi offloading and reduce monthly 3G data bills. Moreover, some delays could present opportunities for intelligent load balancing in the network itself by shifting the bandwidth usage to a less congested timeframe.

Category	3G	Wi-Fi
Availability	100%	45%(Vehicle) / 53%(Walk)
Latency	130 ms	80 ms
Bandwidth	1 - 2 Mbps	2.6 - 5 Mbps

Table 1: Wi-Fi Availability Test in Visiting a Large City

Our work bridges the previous studies with practical offloading support from the transport layer. To facilitate the delayed transfer with as little burden on the application developers as possible, we propose using a TCP-like transport layer that is resilient to network disruptions or failures. In this section, we first check the feasibility of Wi-Fi offloading and the current Wi-Fi offloading practice with popular mobile applications.

2.1 Opportunities for Wi-Fi Offloading

We gauge the viability of Wi-Fi data offloading by measuring the availability, connection time, inter-arrival time, bandwidth and latency in a large city in South Korea. First, we have three people measure the 3G/Wi-Fi availability by visiting popular places in Seoul for 4 days. Second, we draw the similar statistics from previous Wi-Fi/3G usage traces of 97 iPhone users for 18 days [6]. Our measurements are by no means representative, but we believe that they show some sense of feasibility of Wi-Fi-based 3G data offloading in an urban setting.

2.1.1 Wi-Fi Availability in Visiting a Large City

To see how widely Wi-Fi is available in a casual visit to a large city, we measure the Wi-Fi availability, bandwidth and latency in a few popular places in Seoul. We pick four popular places (Gangnam, Myongdong, Insa-dong (all outdoors), and Co-Ex (indoors) in Seoul) where many people visit, and move between them by public transportation. During the 4-day visit, we gathered the data for 27 hours (including 6.7 hours on the subway, and 4.4 hours on a bus). We find that Wi-Fi is available for Internet access for about 45% of the time either on the subway or on a bus (73% on the subway, 5% on a bus), and for 53% of the time while walking around the popular places. Most of these Wi-Fi hotspots are provided by a few ISPs in South Korea. For example, one of the ISPs claims that it has over 87K hotspots nationwide [19].

We also measure the bandwidth and latency from these locations to a server in our lab (about 200 km away from the locations) by implementing a simple Android application. Whenever an Android client meets and connects to a new Wi-Fi AP, it calculates the latency between the client and the server by recording the minimum time it takes to receive a response by a ping request. Next, we measure the bandwidth by transmitting a large file to the server and calculating the total transfer time. Table 1 shows that the Wi-Fi bandwidths are between 2.6 to 5 Mbps on average while those of 3G are from 1 to 2 Mbps. than The average latency of Wi-Fi and 3G are about 80 ms and 130 ms. The Wi-Fi latency looks a bit high presumably because it is used by many people in the areas. Overall,

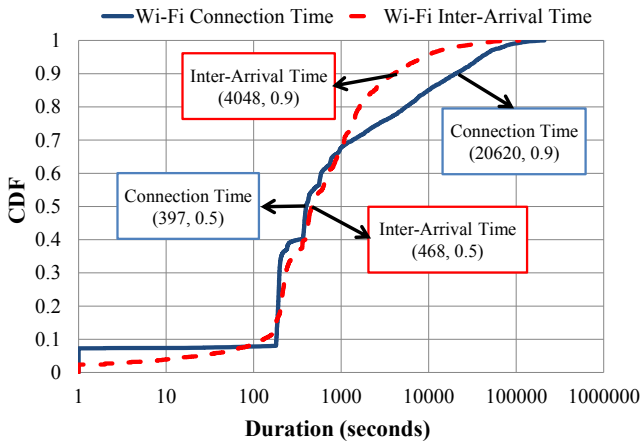


Figure 1: CDF for Wi-Fi Connection and Inter-Arrival Times

our results show that Wi-Fi has larger available bandwidths even in busy places in a large city.

2.1.2 Wi-Fi Availability in Daily Lives

We analyze the Wi-Fi availability in daily lives in a large city. We use the traces of 97 iPhone users who periodically measure the Wi-Fi availability in their daily lives in Seoul for 18 days [6]. We measure the connection and inter-arrival time of Wi-Fi by checking the network status of the client every three minutes. Figure 1 shows the distributions of Wi-Fi connections and inter-arrival times in CDF. The graph shows that about the half of the connection times are less than 6.6 minutes while 10% of them show more than 5.7 hours. The long connection times are mostly for staying at home or at work while small ones indicate transient Wi-Fi availability on the move. Also, about the half of the inter-arrival times are less than 7.8 minutes and 10% of them are over 1.1 hour. This result implies that while there are many short Wi-Fi connections, the inter-arrival times are also small, expecting to meet a new Wi-Fi spot soon.

We find that there are good opportunities for Wi-Fi offloading but in order to maximize the benefit, we need to exploit frequent network disruptions and re-connections to our advantage. In our measurements, we observe that without proper upload or download resumption, the users cannot send or receive a file larger than 120 MB for the half of the Wi-Fi connections and would simply waste the Wi-Fi bandwidth and battery power.

2.2 Mobile Applications in Disruptions

To examine how current mobile applications respond to network disruptions, we analyze the behavior of seven popular Android applications that are downloaded more than 100,000 times from Android market [20] as shown in Table 2. Dropbox [13] provides online storage synchronization, MapDroyd [21] is used to download and store world-wide maps for offline access. Beyondpod [22], Google Listen [23], and Winamp [24] are audio players that can be linked with podcasting services. TubeMate [12] is a video downloader for YouTube.

When mobile devices experience network disconnections during data transmission, Dropbox, MapDroyd, and Winamp stop with a network failure message, and the downloads do not resume even when the network becomes available again. We find that Android Market, Beyondpod, and Google Listen support download resumption by HTTP byte-range queries while TubeMate uses CGI parameter passing. Even though these applications support download

Application	Category	Resumption method
Dropbox	Online storage	Not Supported
MapDroyd	Offline map access	Not Supported
Winamp	Podcast manager	Not Supported
Android Market	App. downloading	HTTP Range Request
Beyondpod	Podcast manager	HTTP Range Request
Google Listen	Podcast manager	HTTP Range Request
TubeMate	YouTube video	CGI Parameter Passing

Table 2: Download Resumption in Popular Mobile Applications

resumption at network disruptions, there is no common library or rules that can be reused for other applications. Also, it is unclear how one supports the streaming contents where the data is generated on the fly.

Even when applications do not implement download resumption, TCP will resume the connection if a disruption clears up within the time for the maximum number of retransmissions of the same segment assuming the IP address does not change. The total time for retransmission before disconnection is recommended to be larger than 100 seconds [25]. We find that the number of maximum retransmissions is 15 on Linux (kernel version 2.6.40), which corresponds to about 17 minutes. This implies that TCP on Linux cannot handle network disruptions longer than 17 minutes even if the IP address does not change. We design DTP to overcome frequent disruptions and IP address change in mobile environments.

3. DESIGN

This section describes the design of DTP, a disruption-tolerant, reliable transport layer protocol. An example is shown in Figure 2. We present the basic protocol, security features, and failure recovery mechanisms at network disruptions.

3.1 Disruption-tolerant Connection

TCP binds the IP addresses and port numbers (or the four tuples) of the two communicating ends on its connection. If any one of them changes, the connection needs to be re-established to resume the data transfer. This implies that the IP address of a TCP connection is used to identify the host location of the network as well as the host itself. This duality of the IP address, however, creates a problem in mobile environments where the host location changes frequently due to host mobility.

In order to maintain a connection despite network disruptions or IP address change, DTP binds a connection to a special identifier called "flow ID". The flow ID is determined at initial connection setup time, and it uniquely identifies the connection on both hosts. The DTP connection persists until either it is explicitly torn down by both ends or the keep-alive duration expires. The keep-alive duration is an estimated connection lifetime set by the application, during which the connection stays on even without the physical network availability. We discuss the details in section 3.2.

If the IP address of a mobile host changes, DTP sends a packet with a new IP address to the other end to initiate the authentication process that proves the ownership of the flow ID. If the other end identifies the connection for the flow ID, both parties can resume data transfer from where it left off. Disruption-tolerant connections bring several advantages. First, it allows the application developers not to worry about frequent network disruptions in the Wi-Fi offloading scenarios. They can assume that the connection is always on until it is done with data transfer. Second, it enables the same

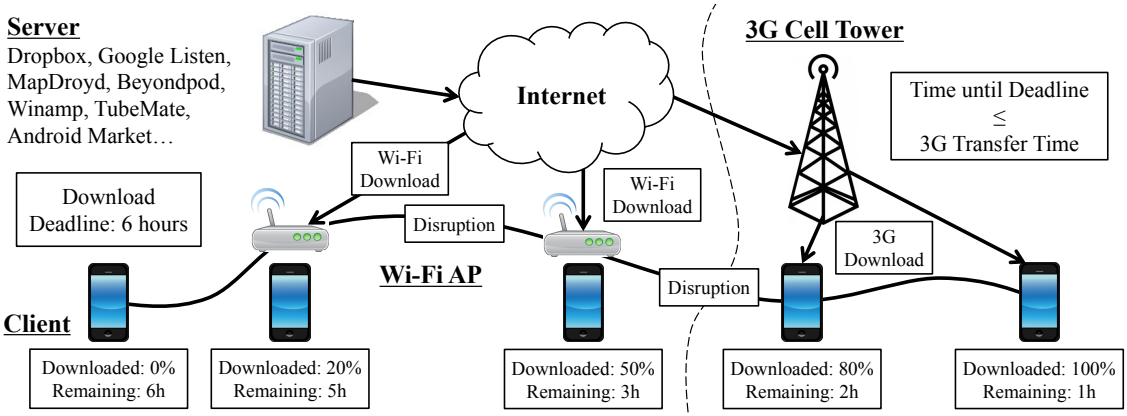


Figure 2: An Example of a Delay/Disruption-Tolerant Network with Wi-Fi Offloading. The client begins downloading from the server with a 6-hour deadline. The file is downloaded through Wi-Fi whenever the client comes in contact with an AP. If the remaining deadline time is less than or equal to the expected 3G transfer time, the client switches from Wi-Fi and downloads through 3G.

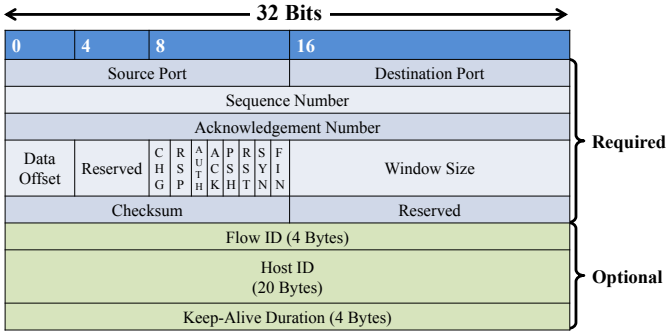


Figure 3: DTP Protocol Header

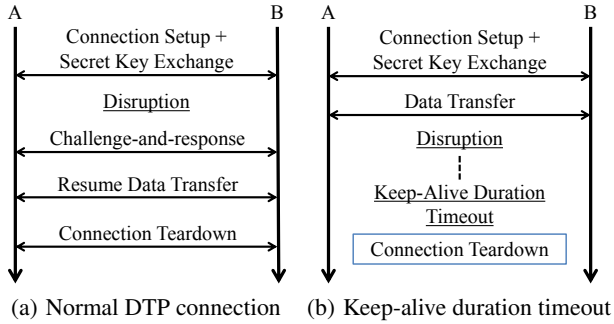


Figure 4: DTP Communication Timeline

connection to switch between Wi-Fi and 3G without notice from the application. It allows seamless offloading of even interactive or real-time data without re-establishing the connection or download resumption by the application. One downside of DTP, however, is in the burden to maintain the connection state even when the host is unavailable. Our prototype DTP implementation requires only 176-byte state information per connection, so even for one million concurrent connections, the system would need less than 200 MB for the states.

3.2 DTP Protocol Header

The DTP protocol header is shown in Figure 3. We borrow most required fields from the TCP header. The CHG, RSP, and AUTH bit flags are used for secret key validation and will be explained in section 3.3.2. In the option fields, we define the flow ID as the last four bytes of the SHA-1 hash of the host ID and the timestamp at the connection creation time (at a microsecond granularity). The host ID is the SHA-1 hash of the host device ID. The host device ID can be any string that uniquely identifies a host during the connection such as International Mobile Equipment Identity (IMEI) of a cell phone or the MAC address of the first interface of a laptop or a PC. The flow ID and the host ID are sent to the remote host at connection initiation, and in the rare case that the same flow ID exists at the remote host for a different connection, the remote host rejects the connection requiring the sender to retry with a new flow ID until there is no conflict.

The keep-alive duration is sent to the remote host as another option field. It is an estimated connection lifetime in seconds set by the application (e.g., dtp_setsockopt() with the KEEP_ALIVE option) before initiating the connection, and can be updated during the connection according to the application's needs. The value is negotiated by the server to limit the number of inactive connections with a very large keep-alive duration. When a host receives a packet with the keep-alive duration option, it either accepts the value by echoing it to the sender or suggests another value in the response packet until both parties agree on the same value. When the keep-alive duration is not used, DTP falls back to the normal TCP behavior and disconnects the connection after 15 retransmissions of the same packet.

3.3 DTP Communication

We describe the persistent data communication with DTP in three stages: connection establishment, data transmission, and teardown as shown in Figure 4(a).

3.3.1 Connection Establishment

To initiate a DTP connection, the sender sends a SYN packet with a flow ID, its host ID, and an optional keep-alive duration value. If the keep-alive duration is missing, the value is initialized to 0. Each host maintains a connection hash table that maps the flow ID to its remote host ID and the four tuples of the connection. If the same flow ID exists for a different connection, the receiving end responds with a RST packet to elicit a different flow ID from the sender. Otherwise, it sends a SYN+ACK with its own host ID

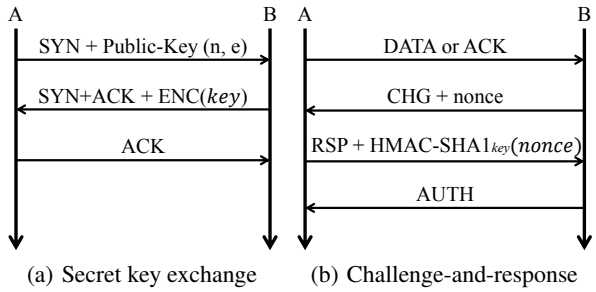


Figure 5: Authentication of hosts using secret key validation

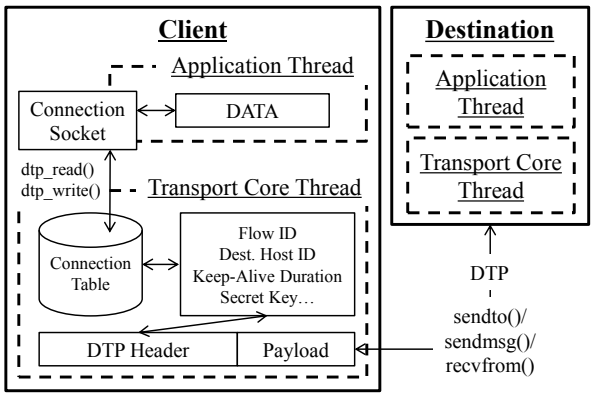


Figure 6: Implementation of DTP

along with the agreed flow ID and an optional keep-alive duration. To prevent connection hijacking, both parties also agree on a shared secret key, which is explained in section 3.3.4.

3.3.2 Data Transmission

After connection establishment, the hosts transfer the data similar to TCP. That is, slow start, sequence numbers and ACK'ing, flow and congestion control work like TCP. At network disruptions, DTP hides the failures from the application and stops data transfer until the network becomes available again. The other end host keeps sending packets until the maximum retransmission threshold is reached. It then either closes the connection or further holds the state without sending any packets, depending on the pre-defined keep-alive duration value. The network availability information can be mostly obtained from the underlying system (e.g., via the BroadcastReceiver package on Android or a netlink socket on Linux) or one can resort to periodic probing with an exponential backoff.

When the network becomes available again, DTP re-synchronizes its connection. The disrupted host sends either a normal data or an ACK packet possibly with a different IP and a port number pair, and the connection resumes after flow ID verification. When a host receives a data/ACK packet whose flow ID does not match the stored IP address and the port number, it responds with an out-of-band challenge packet with the CHG flag on. The challenge packet includes a randomly-generated nonce (8 bytes) in its payload. On receiving it, the other host replies with a response packet that has HMAC-SHA1$_{key}$($nonce$) in its payload with the RSP flag on. After successful verification of the hash, the host sends an authentication packet (with the AUTH flag on), and the connection resumes (Figure 5(b)). If the verification fails, the host sends an RST packet to alert the other host to close and start a new flow.

We note that there are a few cases where the connection resumption may fail. When either host reboots for some reason (e.g., battery outage), it loses all previous connection information. If packets never arrive within the keep-alive duration, the host closes the connection on its end and notifies the application of an error (Figure 4(b)). When a host receives a packet whose flow ID does not exist in its connection table, it responds with a RST packet that has its host ID in the option field. With the host ID, the other end host checks whether the RST packet was sent by the communicating host or by another host that happens to be assigned with the same IP address of the original host after the original host left the network. If a host receives a RST with an unexpected host ID, the connection goes into the wait mode until a resumption packet from the original host arrives or the keep-alive duration expires. When both hosts change their IP addresses during a network disruption, they will end up with closing the connection after the keep-alive duration. However, this case would be uncommon in practice since most connections are between a mobile client and a server whose IP address rarely changes.

3.3.3 Connection Teardown

The connection can be torn down in two ways. The two hosts can explicitly close the connection by exchanging FINs and ACKs like in TCP. If the keep-alive duration expires, the host closes the connection on its end unilaterally. If the application closes the connection during the network disruption, DTP closes the connection on its end but sends a FIN to the other end when the network becomes available again within the keep-alive duration.

3.3.4 Shared Secret Key Exchange

To prevent connection hijacking attacks by a random host, DTP exchanges the secret key at connection setup as shown in Figure 5(a). We use the RSA algorithm here but any asymmetric key cryptographic algorithms can be used instead. The SYN packet includes the public key of the host, (n, e), in its payload, and the other end generates a secret key, encrypts it by the public key, and sends it in the payload of the SYN+ACK packet. This shared key is used to verify the ownership of the flow ID when a connection resumes after IP address change. Since the client side usually initiates the connection, the RSA decryption burden is shifted to the client side in our scheme, alleviating the load at the server. We believe that the additional work done during the secret key exchange or the challenge-and-response does not impose much overhead on the server since a modern CPU core can do more than 3,000 RSA decryptions per second [26] and heavy cryptographic operations can be easily offloaded to GPUs [27].

4. IMPLEMENTATION

In this section, we describe our implementation of the DTP prototype and its API library, which is designed to be compatible to that of TCP for easy migration.

4.1 Architecture

We implement the DTP prototype as a user-level UDP library. We choose the user-level approach for portability and ease of programming, but the more efficient kernel-level implementation would not be too hard. The DTP library spawns a "transport core" thread per application that manages the connection information and processes received packets, and the application thread provides the TCP socket-like functions to the application as shown in Figure 6. Our current prototype is compatible to TCP in terms of functionality: it implements slow start, flow and congestion control, fast re-

```
int dtp_socket(void);
int dtp_bind(int sockfd, const struct sockaddr *addr,
             socklen_t addrlen);
int dtp_connect(int sockfd, const struct sockaddr *addr,
                socklen_t addrlen);
int dtp_listen(int sockfd, int backlog);
int dtp_accept(int sockfd, struct sockaddr *addr,
               socklen_t *addrlen);
ssize_t dtp_read(int fd, void *buf, size_t count);
ssize_t dtp_write(int fd, const void *buf, size_t count);
int dtp_close(int fd);
int dtp_select(int nfds, fd_set *readfds, fd_set *writefds,
               fd_set *exceptfds, struct timeval *timeout);
int dtp_fcntl(int fd, int cmd, ... /* arg */ );
int dtp_getsockopt(int sockfd, int level, int optname,
                   void *optval, socklen_t *optlen);
int dtp_setsockopt(int sockfd, int level, int optname,
                   const void *optval, socklen_t optlen);
uint32_t dtp_getflowid(int sockfd);
```

Figure 7: DTP API Functions

transmit and recovery, timeout and retransmission, delayed ACKs, and so on.

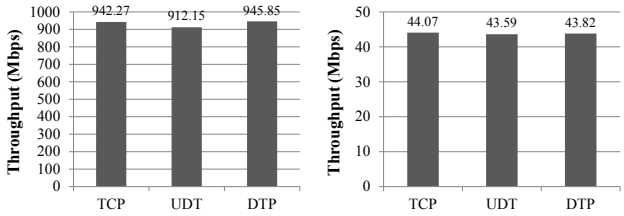
4.2 DTP API Library

One of our implementation goals is to provide easy transition from TCP-based applications. Figure 7 shows the current set of DTP functions that are designed to map to a subset of TCP socket functions. dtp_socket() creates a connection context internally and returns a file descriptor to the application. Using this file descriptor, the application can connect, bind and listen on a port, accept a connection, read and write application data to the other end. A UDP socket is created internally for each connection socket (a socket through which the connection is initiated) and the server listening on a port can accept the connection by creating a UDP socket with a new port number. The mapping between UDP and DTP sockets is managed by the transport core thread. Our current implementation supports event-driven programming with dtp_select(), and we are working on implementing fork(). The number of lines of the current version is 5,283 lines in C.

Our experience with porting existing TCP servers and clients shows that it is straightforward to use the DTP library while it takes small effort to port them. We had one undergraduate student port *wget* [28] to use DTP instead of TCP socket functions. By *grep*ping socket functions and replacing them with the DTP counterparts, he could successfully port it to a DTP version in a couple of hours. It required only 19 lines of code change out of 43,372 lines of the original code. We also port a simple web server to a DTP version with the similar effort. We are currently working on porting the Apache Web server to use DTP, but it requires several function and flag options to be implemented, which is not related to actual network communication.

5. EVALUATION

In this section, we compare the performance of DTP with various protocols in terms of throughput and battery consumption. In our test settings, we use a laptop with a Intel Core i7-2620M processor and 4 GB of physical memory (on linux 2.6.40) and a Nexus-S phone (on Android 2.6.35.7) as clients, and a desktop machine with a Intel Core i7-2600 CPU with 8 GB RAM (on linux 2.6.38-12) as a server.



(a) LAN connection (b) Wi-Fi connection

Figure 8: Large-file Download Tests

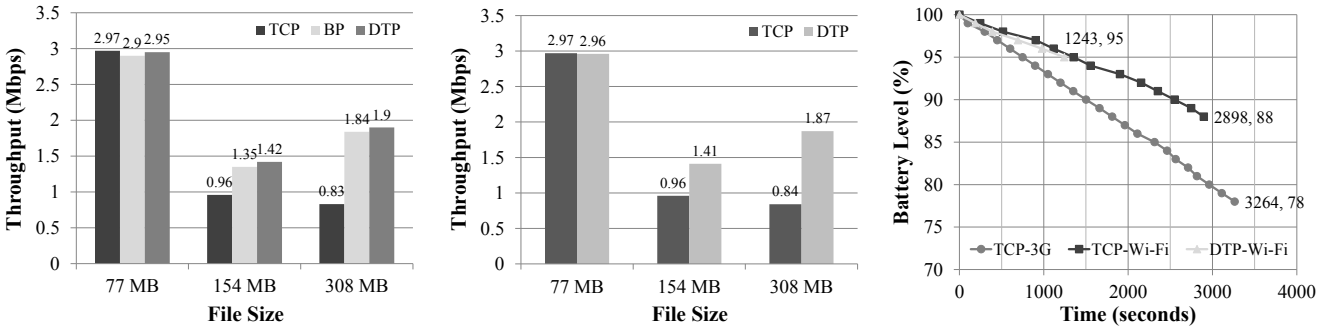
5.1 Microbenchmark

We first measure the base transfer throughputs between a laptop and a desktop server on a 1 Gbps LAN and on a WLAN with an 802.11n Wi-Fi AP. We compare DTP against TCP and UDT, a high-speed reliable data transport protocol based on UDP [29]. In this test, we have the laptop upload a 1 GB file to the server to saturate the LAN connection and a 100MB file for the Wi-Fi connection. As shown in Figure 8, DTP shows a comparable performance to that of TCP both on wired (945.9 Mbps vs. 942.8 Mbps) and wireless LANs (43.82 Mbps vs. 44.07 Mbps). The performance of UDT is also similar to that of TCP and DTP.

5.2 Performance at Network Disruptions

We now compare the DTP performance with that of TCP and the Bundle Protocol (BP), one of the representative DTN protocols [18] while the client changes its IP address when it moves to another Wi-Fi network. The clients are connected to a Wi-Fi AP whose bandwidth we limit to 3 Mbps to simulate our measured results. We use the median connection/disruption time values obtained in Section 2.1 (6.6 minutes and 7.8 minutes). If the content downloading does not complete before the disruption, we increase the next connection time by its median value. We base the file sizes by calculating the average size of the YouTube's top-viewed HD videos for the past one year (each downloaded more than 20 million times) [30], and analyze the impact of network disruptions for the file sizes of 154 MB (average file size), 77 MB (half the average), and 308 MB(double the average).

Figure 9(a) shows the throughputs between the laptop and the server. To allow fast probing of the network availability, we set the maximum backoff time of the probing packet to one second for both DTP and BP. No protocols experience a disruption for downloading the 77 MB file, and the performance is similar among all three protocols. But DTP shows 47.9% and 128.9% better performance than TCP in 154 MB and 308 MB files each. This is because TCP needs to retransmit the entire file after each network disruption, while DTP finishes downloading the files at most in one network disruption. DTP shows 3.3% to 5.2% better performance than BP because BP has extra header overhead of each bundle. While DTP shows only small performance improvement from BP, its resource consumption is much smaller than that of BP since the reference BP implementation holds the entire data in memory to prepare a bundle and creates a new primary block whenever there is a network disruption. Figure 9(b) shows the performance between the phone and the server. Similar to the previous test, DTP shows 46.9% to 122.6% better performance compared with TCP. We do not measure the BP's performance here because the reference implementation does not run on Android due to a lack of support for libraries such as oasys [31]. Another BP implementation, ByteWalla [32] runs on Android, but we find that it does not implement the "Fragment" option (Bundles are split up into multiple constituent bundles



(a) Laptop-to-server Throughput (b) Phone-to-server Throughput (c) Phone Power Consumption

Figure 9: Throughputs and Power Consumption during Network Disruptions

at network disruptions.) [33], and cannot resume data transfer after network disruptions.

5.3 Power Consumption

Figure 9(c) shows the power consumption of Nexus S as we run the tests with the 308 MB file from the previous section. We also measure the power consumption of transferring the same file via TCP using the 3G network *without* disruptions (TCP-3G). TCP-3G shows a rapid decrease in battery power since it consumes more energy during data transfer than Wi-Fi as in [34]. Also, its throughput is the worst (0.78 Mbps) and finishes the last even without network disruptions. We find that DTP-Wi-Fi consumes 58.3% and 77.3% less power compared with that of TCP-Wi-Fi and TCP-3G while it finishes downloading 1,655 and 2,021 seconds earlier.

6. RELATED WORKS

There have been several approaches to support mobility for IP networks. Mobile IP [35] exports a fixed home address through which external hosts communicate regardless of the actual network attachment of the mobile host. When the mobile host leaves its home network, the home agent relays the IP packets arriving at the home address to the *care-of-address* (e.g., real address) of the mobile host. In contrast, DTP does not require a home agent, nor it needs to relay packets, which would produce better packet routing between the two ends.

DTP is similar to Migrate TCP option [36] in that both enable connection reuse for IP address change. But DTP is more friendly to 3G/Wi-Fi offloading environments since it allows the applications to set the disruption delay much larger than the maximum segment lifetime (MSL) DTP bears the similarity with i3 [37] and HIP [38] in that they support mobility by separating the host identity and its network location. Unlike DTP, both require additional infrastructure support such as a DHT network and the DNS.

DTP supports the concept of delay-tolerant networking [9] into Wi-Fi data offloading. Previous DTN protocols such as Bundle Protocol (BP) [18] and Licklider Transmission Protocol (LTP) [15–17] assume more challenged networks with high delays and packet losses whereas DTP is geared towards mostly stable networks but with frequent disruptions. Exploiting the fact, DTP supports reliable transfer on an IP packet level without an additional layer that wraps the content into bundle blocks as in BP. Besides, the current BP works only for the content whose size is already known prior to transmission, whereas DTP allows users to watch a streaming video without interruption even when she moves from a Wi-Fi network to 3G or vice versa. LTP is designed to reliably transfer the data mostly in dedicated networks with very high RTTs (e.g., deep space) and does not consider typical TCP issues such as flow and congestion control in shared networks. For this reason, the current LTP implementation uses pre-defined parameters (e.g., window size) before the communication initiates [39]. In contrast, DTP strives to conform to TCP to be fair to other competing flows, allowing the co-existence of heterogeneous networking devices.

7. DISCUSSION

In this section, we discuss some of the issues that were not addressed in this paper and consider an extended offloading framework for our future work.

State Explosion Attacks: In malicious environments, an attacker can instruct zombie hosts to create many DTP connections with a long keep-alive duration on a target server. While we design DTP to have a very small memory footprint per connection and allow the server to limit the keep-alive duration value to specifically guard against this attack, the application sometimes has to maintain a large buffer per request. One such scenario is that the client sends a large-file request and goes offline immediately afterwards. However, we note that this sort of attack is not unique to DTP but can be launched on any TCP-based servers. One defense approach is for the server to detect suspicious requests by careful resource accounting [40] and dynamically reset the keep-alive durations when it is suspected to be under attack. We plan to explore this issue further in the future.

ISP-driven Offloading Servers: Using DTP, mobile ISPs may further exploit Wi-Fi offloading for efficient network resource utilization. One example is that mobile ISPs provide a DTP cloud storage server that runs an application protocol multiplexer. In this scenario, the multiplexer translates DTP connections from mobile hosts to TCP connections to the target server and vice versa. For instance, a client can send emails to the cloud server using DTP-based SMTP, and the cloud server relays them to the destination using TCP. Or the cloud server can receive and store a podcast video from a TCP-based postcast server and pushes it onto the mobile phone using DTP. This would not only provide an incremental deployment path of DTP, but also allow the mobile ISPs to spread bandwidth consumption across the time axis similar to SmartGrid [41]. We are currently working on a cloud storage server for mobile ISPs.

8. CONCLUSION

While many works have shown the effectiveness of Wi-Fi mobile data offloading, there has not been a practical data delivery mechanism to support it. We propose DTP, a disruption-tolerant reliable transport layer protocol, which allows seamless switching between

3G and Wi-Fi networks on the same connection for mobile applications. We design it to easily migrate existing applications to transparently recover from network disruptions, with little performance degradation from that of TCP. Our evaluation shows that DTP is promising with the great potential to reduce 3G network usage as well as the battery consumption.

9. ACKNOWLEDGEMENTS

We thank anonymous reviewers for their insightful comments. We also thank Yung Yi and Song Chong for lively discussion about the role of the disruption-tolerant transport layer. This work was supported by the KCC (Korea Communications Commission), South Korea, under the R&D program supervised by the KCA (Korea Communications Agency), KCA-2011-11913-05004.

10. REFERENCES

[1] SEO Updates. Mobile vs Desktop Internet Usage Stats 2011, 2011. http://www.seodailyupdates.com/2011/06/mobile-vs-desktop-internet-usage-stats.html.

[2] CISCO. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2010-2015. Technical report, 2011.

[3] J. M. Chapin and W. H. Lehr. Mobile Broadband Growth, Spectrum Scarcity, and Sustainable Competition. In *Proceedings of The 39th Research Conference on Communication, Information and Internet Policy*, 2011.

[4] J. Mitola III and G. Q. Maguire Jr. Cognitive Radio: Making Software Radios More Personal. *IEEE Personal Communications*, 6(4):13–18, 1999.

[5] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting Mobile 3G Using WiFi. In *Proceedings of ACM MobiSys*, 2010.

[6] K. Lee, I. Rhee, J. Lee, S. Chong, and Y. Yi. Mobile Data Offloading: How Much Can WiFi Deliver? In *Proceedings of ACM CoNEXT*, 2010.

[7] J. Scott, P. Hui, J. Crowcroft, and C. Diot. Haggle: A Networking Architecture Designed Around Mobile Users. In *Proceedings of IFIP WONS*, 2006.

[8] A. Chaintreau, P. Hui, J. Crowcroft, C. Diot, R. Gass, and J. Scott. Impact of Human Mobility on the Design of Opportunistic Forwarding Algorithms. In *Proceedings of IEEE INFOCOM*, 2006.

[9] K. Fall. A Delay-Tolerant Network Architecture for Challenged Internets. In *Proceedings of ACM SIGCOMM*, 2003.

[10] Podcast. http://www.apple.com/itunes/podcasts/.

[11] DoggCatcher. http://www.doggcatcher.com/.

[12] TubeMate. http://tubemate.tistory.com/.

[13] Dropbox. https://www.dropbox.com/.

[14] Google/Ipsos OTC MediaCT. The Mobile Movement Study, 2011.

[15] S. Burleigh, M. Ramadas, and S. Farrell. Licklider Transmission Protocol - Motivation. RFC 5325, IETF, 2008.

[16] M. Ramadas, S. Burleigh, and S. Farrell. Licklider Transmission Protocol - Specification. RFC 5326, IETF, 2008.

[17] S. Farrell, M. Ramadas, and S. Burleigh. Licklider Transmission Protocol - Security Extensions. RFC 5327, IETF, 2008.

[18] K. Scott and S. Burleigh. Bundle Protocol Specification. RFC 5050, IETF, 2007.

[19] KT. olleh WiFi zone Finder. http://zone.wifi.olleh.com/en/index.action.

[20] Android Market. https://market.android.com/.

[21] MapDroyd. http://www.mapdroyd.com/.

[22] Beyondpod. http://www.beyondpod.mobi/android/index.htm.

[23] Google Listen. https://market.android.com/details?id=com.google.android.apps.listen/.

[24] Winamp. http://www.winamp.com/android/.

[25] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122, IETF, 1989.

[26] Michael E. Kounavis, Xiaozhu Kang, Ken Grewal, Mathew Eszenyi, Shay Gueron, and David Durham. Encrypting the internet. In *Proceedings of ACM SIGCOMM*, 2010.

[27] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[28] GNU wget. http://www.gnu.org/s/wget/.

[29] Y. Gu and R. L. Grossman. UDT: UDP-based Data Transfer for High-Speed Wide Area Networks. *Computer Networks (Elsevier)*, 51(7), 2007.

[30] YouTube. http://www.youtube.com.

[31] Oasys Documentation Library. http://www.omgeo.com/page/productdocumentation/?var1=oasys/.

[32] R. Yanggratoke, A. Azfar, M. J. P. Marval, and S. Ahmed. Delay Tolerant Network on Android Phones: Implementation Issues and Performance Measurements. *Journal of Communications*, 6, 2011.

[33] V. Cerf, S. Burleigh, L. Torgerson, R.Durst, K. Scott, K. Fall, and H. Weiss. Delay-Tolerant Networking Architecture. RFC 4838, IETF, 2007.

[34] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications. In *Proceedings of ACM Internet Measurement Conference (IMC)*, 2009.

[35] A. Myles, D. Johnson, and C. Perkins. A mobile host protocol supporting route optimization and authentication. *IEEE Journal on Selected Areas in Communications*, 13(5), 1995.

[36] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proceedings of ACM MOBICOM*, pages 155–166, 2000.

[37] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. 2002.

[38] R. Moskowitz and P. Nikander. Host Identity Protocol Architecture. RFC 4423, IETF, 2006.

[39] S. Farrell, V. Cahill, D. Geraghty, I. Humphreys, and P. McDonald. When TCP Breaks: Delay- and Disruption-Tolerant Networking. *IEEE Internet Computing*, 10(4), 2006.

[40] Xiaohu Qie, Ruoming Pang, and Larry Peterson. Defensive Programming: Using an Annotation Toolkit to Build DOS-Resistant Software. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[41] R. E. Brown. Impact of Smart Grid on distribution system design. IEEE, Power and Energy Society general Meeting, 2008.