Cedos: A Network Architecture and Programming Abstraction for Delay-Tolerant Mobile Apps

YoungGyoun Moon, Donghwi Kim, Younghwan Go, Yeongjin Kim, Yung Yi,
Song Chong, *Member, IEEE*, and KyoungSoo Park

***Abstract*— Delay-tolerant Wi-Fi offloading is known to improve overall mobile network bandwidth at low delay and low cost. Yet, in reality, we rarely find mobile apps that fully support opportunistic Wi-Fi access. This is mainly because it is still challenging to develop delay-tolerant mobile apps due to the complexity of handling network disruptions and delays. In this paper, we present Cedos, a practical delay-tolerant mobile network access architecture in which one can easily build a mobile app. Cedos consists of three components. First, it provides a familiar socket API whose semantics conforms to TCP, while the underlying protocol, D^{2}TP, transparently handles network disruptions and delays in mobility. Second, Cedos allows the developers to explicitly exploit delays in mobile apps. App developers can express maximum user-specified delays in content download or use the API for real-time buffer management at opportunistic Wi-Fi usage. Third, for backward compatibility to existing TCP-based servers, Cedos provides D^{2}Prox, a protocol-translation Web proxy. D^{2}Prox allows intermittent connections on the mobile device side, but correctly translates Web transactions with traditional TCP servers. We demonstrate the practicality of Cedos by porting mobile Firefox and VLC video streaming client to using the API. We also implement delay/disruption-tolerant podcast client and run a field study with 50 people for eight weeks. We find that up to 92.4% of the podcast traffic is offloaded to Wi-Fi, and one can watch a streaming video in a moving train while offloading 48% of the content to Wi-Fi without a single pause.**

***Index Terms*— Delay-tolerant networking, Wi-Fi offloading, mobility.**

I. Introduction

Wi-Fi has become the most popular secondary network interface for high-speed mobile Internet access. Many mobile apps support the "Wi-Fi only" mode that helps avoid expensive cellular communication while enjoying high bandwidth and low delay. In addition, cellular ISPs are actively deploying Wi-Fi access points (APs) to further increase the mobile Internet access coverage.

However, current Wi-Fi usage is often *statically* bound to the location of mobile devices. While this "on-the-spot" Wi-Fi offloading is still effective, recent studies suggest that one can further extend the benefit of Wi-Fi access if we allow delay tolerance between network connections [1], [2]. The basic idea is to hold off using cellular networks in the absence of Wi-Fi service and to resume the data transfer when the device meets the next Wi-Fi AP. For example, [2] reveals that Wi-Fi offloading ratio, the fraction of data offloaded to Wi-Fi from cellular networks, could be increased to 93.7% if we allow only one hour of delay between Wi-Fi connections.

Despite the great potential of delay-tolerant Wi-Fi offloading, existing mobile apps rarely support disrupted network operations. This is mainly because the burden of handling network disruption or delay is placed solely on app developers. In reality, it is challenging to build delay-tolerant mobile apps. Existing network stacks are clumsy at addressing network disruptions/delays since they are designed for real-time environments. Moreover, popular servers are unfriendly to delay-tolerant apps that intermittently download the content in multiple networks. We find that many mobile apps do not properly address network switchings and few apps correctly handle even a few minutes of delay between network connections.

Delay tolerance in mobile Internet access requires addressing two mobility events: network disruption and delay between network connections. We refer to network disruption as an event that a mobile device switches from one network to another due to user mobility. Network disruption often results in IP address change, which forces the termination of ongoing TCP connections. To recover from a network disruption event, mobile apps should be programmed to resume from the last downloaded offset or to download the content from the start in a separate connection. Unfortunately, download resumption cannot be a general option since the content could be dynamically generated or the application layer protocol may not support it. Re-downloading from the start is also undesirable since it wastes network bandwidth and power consumption. Exploiting a long delay (*e.g.,* a few hours) between preferred network connections is another major issue. To maximize the Wi-Fi usage opportunity, a mobile app may want to resume network transfer only when the mobile device is within the Wi-Fi coverage. But at the same time,

Manuscript received November 17, 2015; revised April 27, 2016; accepted July 27, 2016; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor A. X. Liu. Date of publication November 29, 2016; date of current version April 14, 2017. This work was supported in part by the ICT Research and Development Program of MSIP/IITP, Korea, under Grant B0101-16-1368, [Development of an NFV-inspired networked switch and an operating system for multi-middlebox services] and in part by the Institute for Information and Communications Technology Promotion Grant Funded by the Korean Government (MSIP) under Grant B0126-16-1078. Part of this work has been presented at ACM MobiSys 2015.

Y. Moon, Y. Go, Y. Kim, Y. Yi, S. Chong, and K. Park are with the School of Electrical Engineering, Korea Advanced Institute of Science & Technology (KAIST), Daejeon 305-701, South Korea (e-mail: ygmoon@kaist.ac.kr; yhwan@kaist.ac.kr; codud3211@kaist.ac.kr; yiyung@kaist.ac.kr; songchong@kaist.ac.kr; kyoungsoo@kaist.ac.kr).

D. Kim is with the School of Computing, Korea Advanced Institute of Science & Technology (KAIST), Daejeon 305-701, South Korea (e-mail: dhkim09@kaist.ac.kr).

This paper has supplementary downloadable material available at http://ieeexplore.ieee.org, provided by the authors. This consists of D^{2}TP packet header format, D^{2}TP socket API, and sample codes.

Digital Object Identifier 10.1109/TNET.2016.2603523

it may want to avoid an indefinite delay but to continue the transfer with a cellular network if the delay exceeds a certain threshold. To the best of our knowledge, no existing solution supports a long delay of network unavailability in the same TCP connection.

In this work, we bridge the gap between theory and practice in delay-tolerant Wi-Fi offloading. We propose Cedos, a practical delay-tolerant mobile network architecture for mobile apps. Cedos eases the development of a mobile Internet app with three salient features. First, it provides a familiar socket-like API whose semantics conforms to TCP in stationary environments. Developers write the code just like they do with the Berkeley socket API, but the underlying protocol below the API, D^{2}TP, transparently handles mobility events such as network disruptions or delays. Second, the Cedos network API explicitly exposes how to handle delays between network connections. Developers may express a deadline and a content size with the API, and D^{2}TP completes the content download in time while exploiting the Wi-Fi network as much as possible. Also, the API provides real-time buffer management such that the transport layer automatically switches between cellular and Wi-Fi networks depending on the buffer availability. This allows to build a streaming application that maximizes opportunistic Wi-Fi usage without risk of being interrupted in the middle of streaming. Third, for backward compatibility, Cedos provides a protocol translation Web proxy, D^{2}Prox. D^{2}Prox manages the delay-tolerant HTTP connections with mobile clients while it downloads the content from the origin Web server using TCP. It also caches popular content to reduce the bandwidth consumption on the server side.

This paper makes two key contributions. First, while many previous works address challenges in delay-tolerant networking and Wi-Fi offloading, Cedos provides the first comprehensive solution to delay-tolerant mobile app development and deployment. There are many works that separate the host identity from its location [3]–[8], but they do not provide network stack abstraction. TCP migrate [9] and Serval [10] support host mobility in a TCP connection, but they do not allow long delays nor expose an API to maximize opportunistic network access. Wiffler [1] exploits the delay in Wi-Fi offloading, but their network API does not conform to the TCP semantics. The goal of Cedos is to popularize delay-tolerant Wi-Fi offloading while significantly reducing the complexity of app development and deployment.

Second, we demonstrate the practicality of Cedos by porting existing mobile apps, VLC streaming client [42] and Mobile Firefox [43], with different network usage patterns. Not surprisingly, we find that large, non-interactive file downloads benefit the most from Cedos, but our real-time buffer management helps maximize Wi-Fi offloading opportunities even for real-time video streaming. In our evaluation with Cedos-vlc, we find that almost half (48%) of the streaming video is delivered via opportunistic Wi-Fi access without a single playback pause on a moving train. The benefit for short, interactive Web flows with Cedos-firefox is modest due to noticeable delays in network switching (*e.g.,* seconds), but we find that porting process is straightforward since the Cedos network API is similar to the Berkeley socket API.

Along with porting existing apps, we build our first delay-tolerant podcast app, ReadyCast [44], to demonstrate the utility of delay-tolerance in real world. Unlike real-time podcast apps, ReadyCast allows "download reservation" with a user-specified deadline, and the content is made ready by the time the user wants. We have conducted a field study with 50 ReadyCast users for 8 weeks, and we find that Cedos does help download most of the podcast contents (92.4%) via opportunistic Wi-Fi access. Moreover, D^{2}TP successfully saves 38% of the data that would have to be re-downloaded due to natural network disruptions from user mobility.

II. Mobile App Behavior at Mobility

In this section, we examine how existing mobile apps react to mobility events. We select 21 popular mobile apps[1] from Google Play store that might benefit from delay-tolerant Wi-Fi access. We report their behaviors on network disruption and during the delay at network unavailability.

Methodology: We install each app on a Nexus 5 smartphone, running Android v4.4. The device is connected to the Internet either via a Wi-Fi AP in our lab or through an LTE connection to a local cellular ISP. To simulate a network disruption event, we had the device use the Wi-Fi AP initially, but (i) forced it to turn off the Wi-Fi interface or (ii) unplugged the power cable of the AP. We note that both events simulate moving the device from a Wi-Fi network to a cellular network, which results in IP address change. For a delay test, we had the device use the Wi-Fi AP, but unplugged the network cable of the AP to simulate network unavailability. We replugged the network cable after 10, 30, 60, 120, 300, 3600 seconds of delay. This process does not change the IP address of the device, but injects a period of Wi-Fi network unavailability. Finally, we examine the apps that support Wi-Fi offloading via "Wi-Fi only" mode by toggling the option during data transfer. Since we do not have access to source code of those mobile apps, we report only qualitative behavior of each app. During each test, we had the device download or upload data whose size is large enough to be interrupted by the mobility event.

Findings: Table I summarizes the results of our experiments. We find that many mobile apps do not handle network disruptions properly while notable exceptions are social networking service (SNS) apps, some media streaming apps, and podcast apps. All apps use standard TCP/IP protocols, so we notice that their TCP connection terminates when the host IP address changes. For graceful operation at network disruption, the developers have to write code to re-initiate a TCP connection to resume the data transfer, but that seems to be often neglected in mobile apps. What is surprising is that some mobile Web browser does not support download resumption even if HTTP supports range queries. In media streaming, app behavior varies - MXPlayer and YouTube apps resume video streaming even after IP address change, but VLC and Google Play Movie get stuck. Interestingly, posting comments in the YouTube app is not disruption-tolerant while its video streaming is. The disparity comes from the fact that the developers have

[1]We use the latest version of each app as of November 2014.

TABLE I

BEHAVIOR OF POPULAR MOBILE APPS AT MOBILITY EVENTS

Type	App	Disruption Test		Delay Test		Opportunistic Wi-Fi Offloading
		Task	Result	Task	Result	
SNS	Twitter	Upload a photo	✓ Resume	Upload a photo	✓ Resume	✗ No consideration
	Facebook				✗ Fail if $D > 5$ min	
Podcast	Podcast Addict	Download 5MB or larger podcast file	✓ Resume	Download 5MB or larger podcast file	✓ Resume	✗ "Wi-Fi only" (W1, W3)
	BeyondPod				✗ Fail if $D > 2$ min	✗ "Wi-Fi only" (W2)
	Podcast Republic				✗ Fail if $D > 1$ min	✗ "Wi-Fi only" (W3)
Media streaming	YouTube	Stream video Post comment	✓ Resume ✗ Fail	Stream video	✗ Playback interrupts	✗ No consideration for non-HD movies
	TuneIn Radio	Stream audio Tune into channel	✓ Resume ✗ Fail	Stream audio	✗ Playback interrupts	✗ No consideration
	MXPlayer	Stream video	✓ Resume	Stream video	✗ Playback interrupts	
	Google Play Movie		✗ Fail			
	VLC					
Web browsing	Chrome	Download 100MB file Load static web page	✗ Fail ✗ Fail	Download 100MB file	✗ Fail if $D > 5$ min	✗ No consideration
	OperaMini	Download 100MB file Load static web page	✓ Resume ✗ Fail		✗ Fail if $D > 1$ min	
Online shopping	eBay	Load page	✗ Fail	Load page	✗ Fail if $D > 1$ min	✗ No consideration
	Amazon	Load page Play demo song	✗ Fail ✗ Fail	Load page Play demo song	✗ Fail if $D > 30$ sec ✗ Fail	
	Google Play Book	Load ebook	✗ Fail	Load ebook	✗ Fail if $D > 1$ min	✗ "Wi-Fi only" (W1, W3)
P2P	uTorrent Android	Download 100MB file	✓ Resume	Download 100MB file	✓ Resume	✗ "Wi-Fi only" (W1, W3)
Remote desktop	TeamViewer	Connect	✗ Fail	Remote desktop Transfer 100MB file	✓ Resume ✓ Resume	✗ No consideration
VoIP	Viber	Video chat	✓ Resume	Video chat	✗ Fail if $D > 30$ sec	- N/A
	Skype	Voice Chat Login	✓ Resume ✗ Fail	Voice chat	✗ Fail if $D > 1$ min	
Online game	Clash of Clans	Play	✗ Fail	Play	✗ Play interrupts	- N/A
	Fifa Online 3 M	Play	✗ Fail	Play	✗ Play interrupts	

* 'Fail' means no download resumption nor re-downloading. D refers to delay.

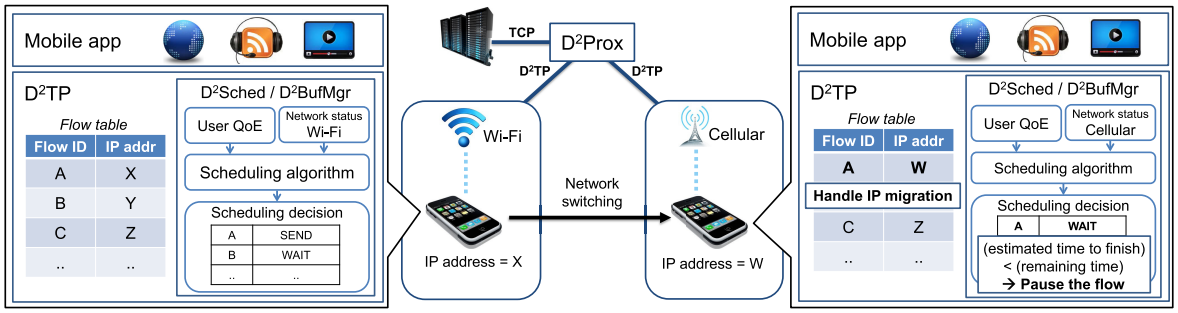


Fig. 1. Cedos delayed data offloading architecture. D²TP handles both network disruptions and long delays. D²Sched schedules the flows for delayed Wi-Fi offloading (e.g., it uses a Wi-Fi connection as much as possible but sometimes switches to an LTE network to finish in time).

to handle each case differently without the support from the transport layer. In contrast, all three podcast apps handle short network disruptions well. We think that podcast app developers are better-prepared for frequent network disruptions at large file download, and handle this case well.

The behavior for the delay test, however, is more disappointing. Except Twitter and Podcast Addict, all apps fail to resume data transfer in a few minutes of network unavailability. We also find that all media streaming apps pause the playback during the delay. Since none of them support the "Wi-Fi only" mode, a desirable action would be to automatically switch to the cellular interface and to resume streaming if Wi-Fi access becomes unavailable. Twitter and Podcast Addict handle both network disruptions and delays well, but they do not support download deadline as discussed shortly.

Lastly, we have investigated if the apps support opportunistic Wi-Fi offloading. We find that some apps allow the "Wi-Fi only" mode, but none of them support any notion of download deadline. Without a deadline, a download in the "Wi-fi only" mode would wait indefinitely when there is no Wi-Fi connectivity. We further analyze how the apps behave with the "Wi-Fi only" option. We try turning on the option while downloading through an LTE interface and try turning it off while downloading via LTE network. No apps that support the "Wi-Fi only" mode handle these two cases correctly. They either continue with the cellular interface (W1) or end up with a download failure in the former case (W2). Or they do not resume the transfer by switching to the cellular interface in the latter case (W3).

In summary, we find that many popular mobile apps neither properly handle mobility events nor their behavior is consistent across app types. This is because the current network API assumes real-time operation and app developers have to write the action on their own. The goal of our work is to ease their burden by transparently handling the mobility events in the transport layer while providing the flexible knob to control the behavior with a familiar API.

III. DESIGN

In this section, we present the design of Cedos, a practical delay-tolerant networking architecture for mobile apps. Cedos transparently addresses mobility events in a new transport layer while it allows flexible control between opportunistic Wi-Fi and cellular network usage.

A. Overall Architecture

Figure 1 shows the overall architecture of Cedos. A Cedos-based mobile app initiates a D²TP connection to D²Prox which relays downloading content from a TCP server to

the app. D^{2}TP is a TCP-like transport layer protocol, but it transparently masks network disruptions and delays. D^{2}Prox works as a protocol translator, allowing client-side mobility while it supports backward compatibility to TCP-based servers. We assume that a mobile device can move around, and transfer data intermittently in multiple networks while the server is at a fixed location. Each D^{2}TP connection is characterized by a maximum-allowed delay, deadline, and a preferred network interface (*e.g.,* Wi-Fi). Until the deadline, D^{2}TP strives to transfer as much data as possible through the preferred network, even by intentionally delaying the transmission if the preferred network is unavailable.

For connections without a deadline or near deadline expiration, D^{2}TP uses any available interface to complete the data transfer. D^{2}Sched analyzes the current network status (available bandwidth and interfaces) together with a deadline and a content size, and decides if a flow should transfer data, wait for the preferred interface, or switch to a faster network. For dynamic content whose size is unknown (*e.g.,* video stream), D^{2}BufMgr calculates an appropriate deadline and size by analyzing the minimum throughput required to maintain the app's Quality of Experience (QoE) (*e.g.,* bitrate), and chooses how a flow should transfer data similar to D^{2}Sched.

B. D^{2}TP: Transport Layer for Mobile Apps

D^{2}TP is a transport layer protocol for mobile apps, providing TCP-like, reliable data transfer in stationary environments but it hides network disruptions and allows delays when a mobile device is on the move. We choose to take the transport layer approach, hiding the mobility events from the application layer. This decision presents two advantages. First, it frees the mobile app developers from directly dealing with network disruptions or delays while providing a flexible knob to control the deadline. The rationale is that the developers need to focus on the core program logic rather than to keep track of the latest offset of an interrupted content or whether a network interface is on or off. Second, having this mechanism inside the transport layer allows providing more information as to delay-induced transmission with D^{2}Sched and D^{2}BufMgr, explained later. Maintaining the amount of data that has been transferred through Wi-Fi and cellular interfaces allows a more informed decision whether to transmit or not at any given time.

The key enabler for D^{2}TP disruption tolerance is in the separation of a connection from its network attachment as in [3], [5], and [8]. Our contribution here is to implement the mechanism in a purely end-to-end fashion and to make it easy to use in real mobile networks. Specifically, each D^{2}TP connection is identified by a persistent flow id and a host id that do not change in the course of mobility. When a mobile device moves to another network, it resumes the connection with the flow and host ids from where it left off. One downside may be the overhead of keeping track of many idle connections at the D^{2}TP server, but we keep the connection metadata small enough to maintain as many as one million concurrent flows for less than 1 GB memory.

1) Connection setup and teardown: D^{2}TP is similar to TCP except it requires an explicit connection resumption process when the network is available again after disruption. Connection teardown may occur explicitly or implicitly depending on the user-defined deadline.

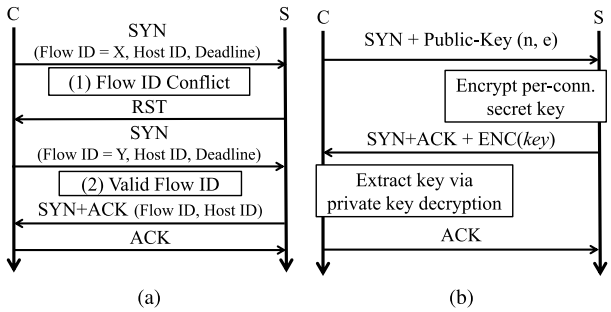


Fig. 2. Initialization of a D^{2}TP connection. (a) Flow creation. (b) Secret key sharing.

Connection Setup: Figure 2a shows how a mobile client establishes a D^{2}TP connection with a server. The client first sends a SYN packet, but unlike TCP, the SYN packet must include a flow id and a client-side host id as its option fields. The host id is defined as a 20-byte SHA-1 hash of the MAC address of the cellular interface. The flow id is the least significant 4 bytes of the SHA-1 hash of the host id and a microsecond-granularity timestamp at connection setup. Since the purpose of the id pair is to identify the connection between the client and the server, the pair needs to be unique only on the two machines during connection lifetime. The SYN packet may include a deadline specified in seconds, as a 4-byte option field. When it is missing, the connection falls back to a normal TCP connection.

If the server already holds a D^{2}TP connection with the same flow id, it responds with a RST to ask for retrying with a new flow id. Or, the server replies with a SYN+ACK packet containing the server-side flow and host ids, and the client establishes the connection with an ACK. Besides a regular three-way handshake, the D^{2}TP server and client need to agree on a per-connection secret key to authenticate each other when the connection is disrupted and resumed later. Figure 2b shows the key exchange process. We employ an asymmetric key cryptographic algorithm (*e.g.,* RSA) and have the client send its public key (n, e) in the payload of the SYN packet. The server then generates a secret key, encrypts it with the client's public key, and sends it in the payload of the SYN+ACK packet. The client finally decrypts the message with its private key and stores the extracted secret key into the flow's state table for future use.

Connection Teardown: A D^{2}TP connection is closed in two ways. First, when both hosts complete the data transfer, they explicitly close the connection by exchanging a FIN. Second, if an application closes the connection during network unavailability, the D^{2}TP layer waits until the network becomes available again, and does a normal closure. If the deadline expires before the device meets an available network, the D^{2}TP connection terminates itself without notifying the other. This should not be a problem since the other party would close the connection as well. The deadline can be updated by the application at any time during the connection to reflect the effective network bandwidth and actual transferred data size.

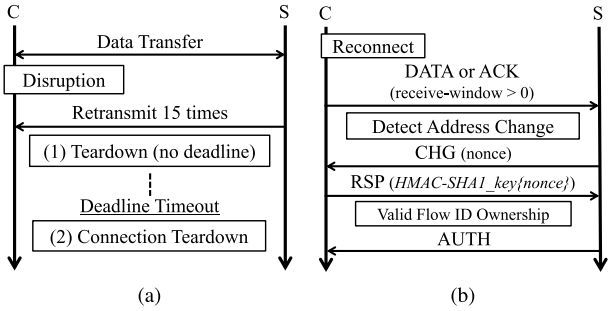


Fig. 3. Flow handling in mobile environments. (a) Teardown at disruption. (b) Challenge-and-response.

2) Data Transfer in Mobile Environments: Once a connection is established, the client and the server exchange data as in a normal TCP connection. On network disruption, connection is resumed with a new IP address after verifying the ownership of the connection.

Connection Resumption: If a client-side IP address changes due to network switching, the client can resume the connection by authenticating itself to the server to prevent connection hijacking. It first piggybacks flow and host ids as options to a normal data or ACK packet. Then, the server detects the client-side IP address change, and verifies the connection ownership through a challenge-and-response protocol shown in Figure 3b. For this protocol, we define three special bit flags (CHG, RSP, and AUTH) in D^{2}TP packet header (see Appendix A in the supplementary file). The server first sends a challenge packet by setting the CHG bit flag on with an 8-byte random nonce in the payload. Then, the client calculates $HMAC-SHA1_{key}(nonce)$, using the shared key at connection setup, and sends it back with the RSP bit flag on. Once the hash is verified, the server responds with an authentication packet with the AUTH flag on, which gives an OK sign that the client can resume the data transfer. On verification failure, the server sends an RST packet to notify the client to start a new connection.

C. D^{2}TP Flow Management

Cedos achieves efficient mobile network usage by avoiding the cellular data transfer whenever an app allows a delay. We describe how a D^{2}TP flow is managed to determine the scheduling action: wait, transfer data, or switch interface for minimum QoE guarantee.

1) Deadline-Based Flow Management: The core enabler for managing an efficient D^{2}TP flow data transfer is the deadline, set explicitly by Cedos socket API. Basically, the deadline is examined to decide whether the D^{2}TP flow should transfer data through the attached network for efficiency (Wi-Fi offloading), or should change the interface for reliability (deadline guarantee). For this, we collect information of currently attached interface, bandwidth for both networks (last measured for unconnected network), flow deadline and content size. When the mobile client moves out of Wi-Fi, the remaining deadline and time duration to transfer remaining content through a cellular network are calculated for flow scheduling. Additionally, the remaining deadline is periodically monitored to determine whether the current network throughput is large enough to transfer the content in time. Detailed algorithm on how D^{2}TP flows are scheduled will be explained in Section III-D.

2) Exploiting Buffer for Flow Management: Unfortunately, it is difficult to directly apply deadline-based flow management for dynamic content apps (*e.g.,* streaming) as there is no deterministic way to identify when the data transfer would end. A straightforward solution would be to periodically adjust the deadline and content size to match the minimal bitrate required. However, this can frequently interrupt streaming as the flow will not begin data transfer until the deadline is close. Reversely, setting too small deadlines would result in frequent CPU wakeups for deadline calculation, consuming much of mobile device's limited battery resources. Therefore, we propose an alternate method to calculate an appropriate deadline and content size, to be later used for flow scheduling.

D^{2}BufMgr is a flow manager for dynamic content that exploits a popular design semantics that many streaming apps follow for efficient playback: buffer that holds to-be-played content. The goal of D^{2}BufMgr is to delay data transfer in a cellular network by consuming already buffered data only until the point that would result in interruption. For this, we introduce a *low* threshold, which represents the remaining buffer point where D^{2}TP flow should begin transferring data through a cellular network, assuming that either Wi-Fi is unavailable or its signal strength is too weak for continuous streaming. The deadline is thus determined by the estimated time that it takes to actually download the data via cellular network (= hardware interface switching time + D^{2}TP's challenge-and-response message exchange time + first data packet arrival time) while the content size is minimum data required to be transferred during that time according to its bitrate.

To minimize cellular traffic usage, the device must return to a Wi-Fi network when it has buffered enough data with a cellular network. To this end, D^{2}BufMgr introduces a *high* threshold, that determines when the device stops using a cellular network by either switching to Wi-Fi if it is available or simply pausing the data transfer. The *high* threshold can be calculated similarly, which should be set large enough to begin downloading the data via Wi-Fi before reaching the *low* threshold. During cellular traffic usage, D^{2}BufMgr resets the deadline and content size to match the *high* threshold. Reversely, when the buffer reaches the *high* threshold, D^{2}BufMgr swaps the values back to guarantee the *low* threshold.

D. D^{2}TP Flow Scheduling

When a mobile device opens a D^{2}TP flow, a flow scheduler, called *D^{2}Sched*, uses the values set from flow management to schedule the flow such that the flow should be idle or active in transmission and which interface it should use if it is active. In case of multiple flows, D^{2}Sched schedules flows by considering the contention between shared mobile resources (*e.g.,* bandwidth).

Scheduling Algorithm: The scheduling algorithm of D^{2}Sched is described in Algorithm 1. Before explanation, we introduce several notations for ease of exposition. At each scheduling epoch with m D^{2}TP flows, each D^{2}TP

Algorithm 1 D^{2}Sched Algorithm

At each scheduling epoch,
Input: $(f_i, T_i : i = 1, \ldots, m)$, r^c, and r^w
Output: I (the interface to use) and $i^\star$ (the flow to schedule)

1: At each scheduling epoch,
2: For each flow i, let $\tau_i = \sum_{j=1}^{i} \lceil f_j / r^c \rceil$.
3: **if** $\tau_i \leq T_i$, for all flows $i = 1, \ldots, m$,
4: **if** Wi-Fi is unavailable
5: $I = \texttt{none},\ i^\star = 0,$
6: **else** $I = \texttt{Wi-Fi},\ i^\star = 1.$
7: **end if**
8: **else**
9: $I = \arg\max_{n=\{c,w\}} r^n,\ i^\star = 1.$
10: **end if**

flow has remaining deadline of T_i, and content size of f_i where $i \in \{1, \ldots, m\}$ respectively. We assume that T_i is indexed in the increasing order of flow index i, *i.e.,* $T_1 \leq T_2 \leq ... \leq T_m$. We also let r^c and r^w be the estimated throughput (at this scheduling epoch) of cellular and Wi-Fi, respectively ($r^w = 0$ if the device has no Wi-Fi connection), for which we will explain how to measure them shortly.

In *line 1*, we first compute τ_i for each flow i, which corresponds to the time to finish flow i's data delivery, if the flows were served in the increasing order of their remaining deadlines. We call τ_i as flow i's *potential completion time*. Then, we check whether all flows' potential completion times are smaller than their deadlines (*line 2*), which means that the system is not in the "urgent" situation, *i.e.,* all flows have sufficient time until their deadlines expire. In this case, if the device has no Wi-Fi connection, it just waits and no flow is scheduled (*line 4*), but schedules the flow with the shortest remaining deadline to use Wi-Fi (*line 5*) otherwise. However, if we are in the urgent situation, where there exists a flow whose remaining deadline is less than its estimated completion time, we perform the Earliest Deadline First (EDF) scheduling by serving it via the network interface that would result in a larger expected throughput (*line 8*).

Scheduling Epochs and Throughput Estimation: Scheduling in Figure 1 is periodically carried out every N seconds with no environmental change (*e.g.,* Wi-Fi availability, flow configuration). In each scheduling epoch, r^c or r^w is updated by the average throughput over the past N seconds. Clearly, N should be appropriately chosen, such that it reacts to time-varying network conditions fast as well as it avoids too frequent switchings between Wi-Fi and cellular networks. In our real-world experiment, we choose it to be 1 second, considering both interface switching time and energy consumption. A smarter alternative is to keep a history of Wi-Fi access and to adjust N according to the access pattern [11], [12], but we choose simplicity and fast adaptation at the cost of a slight overhead. Scheduling is also launched in an asynchronous manner to the change of flow configuration and Wi-Fi availability. In particular, in case a mobile device acquires a new Wi-Fi connection, it first spends N seconds without active transmission to estimate r^w.

Design Rationales: We design D^{2}Sched by the following rationales. First, D^{2}Sched naturally enables a single D^{2}TP flow to be assigned a low priority over other real-time TCP flows. This is because a *sequential* scheduling of serving only a single D^{2}TP flow inside D^{2}Sched leads our estimated data rate r^c or r^w, which approximately corresponds to the throughput of a normal TCP flow, to be divided over m D^{2}TP flows. This design comes from the rationale that D^{2}TP flows are less interactive large flows (*e.g.,* backup of data in smartphones or buffering of streaming movie) which should be served with lower priority than other real-time TCP flows. Second, a sequential scheduling of a single flow in an EDF manner is motivated by the following rationales that *(i)* an urgent flow has less opportunities of data transfer than others, and *(ii)* serving multiple flows concurrently generates frequent switching between flows with active transmission, causing overheads such as switching delay and small data rates due to the need of TCP's ramping-up time (*i.e.,* slow-start). We note that the sequential flow scheduling does not hurt the performance of offloading efficiency when the scheduling interval is short enough and the achievable sum throughput is independent of the number of activated flows; D^{2}Sched is an optimal scheduling policy in terms of offloading ratio for a fixed Wi-Fi and cellular throughputs [13]. In spite of time-varying nature of wireless connections, D^{2}Sched is expected to work in an highly efficient way, because Cedos targets at long flows, and mobile users are typically in indoor Wi-Fi networks, where data rate variations are reasonably small [14]. For simplicity, we prioritize a single flow with the earliest deadline even when multiple D2TP apps are running, which works well in practice. We leave more sophisticated schemes such as priority queuing and bandwidth isolation to future work. Third, D^{2}Sched tries to increase the offloading ratio by delaying flows as much as possible, *i.e.,* data transfer is even paused when Wi-Fi is unavailable and remaining deadlines are large.

E. Power Efficient CPU/Wi-Fi Wakeup

One practical concern when enabling flow scheduling in mobile devices is detecting environmental changes at idle device states. Current mobile OS platforms such as Android minimize battery consumption when a mobile device becomes idle by entering the CPU sleep or Wi-Fi "no-attach" mode. In this mode, the CPU is completely switched off until any external input event arrives, and all D^{2}TP flows in the background would stop even if Wi-Fi networks are available for offloading. What is worse is that the device will not retry scanning Wi-Fi APs during the sleep mode, and it would miss potential Wi-Fi opportunities. We thus require a mechanism to transparently wake up CPU and scan for Wi-Fi without user intervention.

One naïve workaround is to prevent the CPU and Wi-Fi interface from turning off by acquiring locks on them in the sleep mode, but it is impractical since it would quickly drain the battery. Instead, we decide to implement a D^{2}TP flow aware lock, called *D-Lock* that periodically locks CPU and Wi-Fi resources. If a preferred network is available, and there

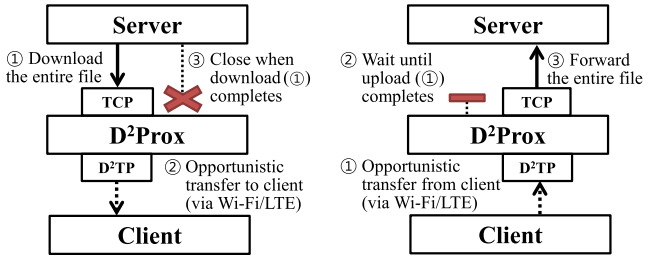


Fig. 4. Protocol translation in D^{2}Prox. (a) D^{2}Prox on download case. (b) D^{2}Prox on upload case.

are D^{2}TP flows that want to use the network, D-Lock keeps the CPU on even if the device becomes idle. When the Wi-Fi network becomes unavailable, it allows the device to go to the sleep mode, but it periodically wakes up the CPU to probe the availability of Wi-Fi APs. The developers only need to hold a D-Lock during the D^{2}TP connection, and it automatically handles periodic scanning of APs at minimal power usage. Through real-world experiments, we find that locking with a period of 2 minutes is enough to detect most environmental changes with minimal power, shown in Section V-B.2.

F. D^{2}Prox: Protocol Translation Proxy

Upgrading the existing infrastructure (*e.g.,* servers, middleboxes) to support D^{2}TP is costly and time-consuming. Instead, we propose placing a network-embedded Web caching proxy, D^{2}Prox, which enables D^{2}TP data transfers between a mobile client and an unmodified server. It basically hides the client-side mobility from the server by communicating with the client in D^{2}TP, but connects to the server using TCP. Note that D^{2}Prox operates as a non-transparent proxy that app developers or users can choose, depending on the characteristics of the apps. Although D^{2}Prox can sit anywhere in the Internet, we envision that it is located near a cellular ISP's core network to curb the latency stretch between an origin server and a client on a cellular interface.

Figure 4 shows the overall architecture of D^{2}Prox. When a new D^{2}TP connection is requested, D^{2}Prox creates an entry in the flow metadata table mapped by the flow id. Each entry consists of client/server IP addresses, a requested URL, and a connection deadline. Since TCP does not allow delay/disruptions, D^{2}Prox only connects to the server when a full data transfer is possible in a single TCP connection. For downlink, D^{2}Prox simply downloads the entire content from the server, caches it in memory or disk, and forwards it to the client if the client is attached to a network. For uplink transfer (*e.g.,* HTTP POST), D^{2}Prox merges all of client's partial data from multiple physical connections, and forwards it to the server. D^{2}Prox removes the flow entry when the connection is explicitly closed or when the deadline expires but the client is out of reach.

Since D^{2}Prox maintains the flow metadata table in memory, a large number of connections with long deadlines could lead to memory pressure. We mitigate this problem by periodically flushing idle flow entries with long deadlines to disk. We believe this approach is reasonable in normal situations since such idle connections will be likely to stay paused to minimize the cellular traffic usage. However, we understand this is not a solution to a Denial-of-Service (DoS) attack on D^{2}Prox, which would require more careful thoughts. One mitigation approach would be to limit the number of concurrent D^{2}Prox flows per mobile device. If a cellular ISP employs D^{2}Prox, it could further require client authentication before using D^{2}Prox.

IV. Implementation

We describe the implementation of Cedos components. We show the practicality of Cedos by porting two existing mobile apps (Mobile Firefox, VLC) to run on Cedos, and implement a delay-tolerant podcast client, ReadyCast, for real-world deployment.

A. D^{2}TP

We implement D^{2}TP in the user level on top of UDP for easy portability and programming convenience. We implement all TCP core features such as slow start, flow and congestion control, fast retransmit and recovery, timeout and retransmission, delayed ACKs, selective ACKs and so on. These are implemented as a separate thread for each D^{2}TP application.

1) D^{2}TP Socket API: D^{2}TP supports ease of portability by providing a socket API where each function maps to a BSD socket function by attaching the D^{2}TP prefix, `d2tp_`. For example, `d2tp_socket()` returns a UDP socket while its associated context information is maintained in the D^{2}TP thread. Applications use this socket to bind, connect, listen on a port, and send or receive data with D^{2}Prox or any D^{2}TP-based applications. For event-driven programming, we support `d2tp_select()` and have it handle events for both D^{2}TP and TCP sockets. Since D^{2}TP socket is implemented as a UDP socket, `d2tp_select()` can simply use `select()` for event notification. Besides regular socket functions, we have special socket option support with `d2tp_setsockopt()` that allows setting a flow deadline (`D2TP_SO_DEADLINE`), a flow size (`D2TP_SO_ESIZE`), and *high* and *low* buffer thresholds for flow scheduling (`D2TP_SO_RCVBUFTH`). One can find the D^{2}TP socket API and its code examples in the online supplementary material.

2) Detecting Network Interface Availability: To pause or resume the data transfer, the D^{2}TP layer must detect the change in the network availability. At start, the D^{2}TP layer reads the current status of network interfaces in the routing table at */proc/net/route*. Then, it monitors a netlink socket to be notified of any status change events of the network interfaces. If D^{2}Sched detects any D^{2}TP flows whose deadlines are about to expire but the quality of a Wi-Fi network is suspected too poor to meet their deadlines, the D^{2}TP layer switches to using the cellular network interface by modifying routing table entry.

3) D-Lock: We implement D-Lock by extending Android WakeLock [40] using AlarmClock [41]. If there is any active flow, D-Lock acquires WakeLock to prevent CPU from entering sleep mode. If there is no active flow, D-lock releases WakeLock and sets AlarmClock to wake up CPU for periodic Wi-Fi scanning. For every scanning period, D-Lock checks if

TABLE II
TOTAL IMPLEMENTED/MODIFIED LINES OF CODE

Implementation	Total (LoC)	Modified (LoC)
D^{2}TP thread and API	11,166	all
D^{2}Prox (nginx)	129,340	55
Mobile Firefox	8,851,611	123
VLC	3,733,651	61
ReadyCast	9,868	all

TABLE III
THROUGHPUT/CPU USAGE DURING WIRELESS DATA TRANSFER

	Galaxy S3		Nexus 5	
Protocol	**Throughput**	**CPU**	**Throughput**	**CPU**
D^{2}TP	89.1 Mbps	16%	90.6 Mbps	11%
UDT	89.3 Mbps	25%	90.5 Mbps	14%
TCP	49.2 Mbps	7%	91.3 Mbps	8%
IBR-DTN	45.1 Mbps	15%	87.0 Mbps	18%

D^{2}TP has any flow that needs to start before the next scan. If so, D-Lock sets another AlarmClock to trigger the D^{2}TP layer to continue the data transfer to meet the deadline.

B. D^{2}Prox

We base our D^{2}Prox implementation on a popular open-source Web server, nginx (v1.2.6), by porting it to use D^{2}TP socket functions. D^{2}Prox supports memory and disk caching, and transfers data to/from a server with normal TCP socket functions when it is able to send/receive a full content in a single connection. Porting is mostly straightforward by replacing existing socket functions with D^{2}TP API, and the total number of modified lines is only 55 out of 129K lines of code as shown in Table II. We have stress tested it to confirm correct delay-tolerant operation as well as Web caching.

C. Real-world Applications

To show the applicability of Cedos to real-world applications, we port two existing applications and implement a delay-tolerant podcast downloader. The number of modified lines of code is shown in Table II.

Mobile Firefox: Since a Web browser is one of the most popular applications, we port mobile Firefox (Fennec [43]) to use D^{2}TP and retrieve web pages via D^{2}Prox. Original Fennec is built on top of NetScape Portable Runtime (NSPR) [45], which provides a platform-independent abstraction through a unified library API. We thus identify all network-related functions in Fennec and NSPR, and change them to use D^{2}TP API. With only 123 lines of code modification, Mobile Firefox enjoys seamless web browsing even after network disruptions. While it works well with large file download, we notice a few seconds of network interface switching delay at network disruptions when we browse regular Web sites; due to this reason, we do not evaluate it further in the next section.

VLC: To demonstrate that streaming media players could also benefit from Cedos, we port VLC [42] (v0.2.0), a popular open-source video player. We modify only 61 lines out of 3.7M lines of code to download streaming video via D^{2}Prox. VLC exploits D^{2}BufMgr in D^{2}TP by setting an appropriate deadline, and delays cellular data transfer if possible to offload to Wi-Fi. We find that by maintaining D^{2}BufMgr's buffer thresholds large enough to buffer drainage during network switching, VLC enjoys a constant video streaming without a single interruption (or buffer underrun) even in a moving train, further explained in next section.

Podcast Client: Since there exists no deadline-based mobile app in the market, we have built our own delay-tolerant publish-and-subscribe podcast downloader, ReadyCast [44], and have registered it with Google Play store. ReadyCast supports subscription to any podcast feed published in the Internet. As shown in Fig. 5, it allows the user to set the deadline of a podcast content download, benefiting from the D^{2}TP layer for transparent delay management. ReadyCast also uses D-Lock to minimize the power consumption at idle states while maximizing the offloading opportunities.

V. EVALUATION

We evaluate Cedos in two ways. First, we evaluate the basic functionalities of Cedos by measuring the data transfer throughput and bandwidth fairness, network switching delay, effect of flow scheduling, power consumption at idle states, and D^{2}Prox scalability. Second, we test if Cedos-based systems effectively offload cellular traffic to Wi-Fi networks in real-world applications.

A. Microbenchmark

We first measure the basic performance of Cedos components. We use Galaxy S3 (Android 4.1.2 and Linux kernel 3.0.31) with an 1.4 GHz quadcore CPU and 2 GB RAM or Nexus 5 (Android 4.4.2 with Linux kernel 3.4.0) with a 2.3 GHz quadcore CPU and 2 GB RAM as a client, and use a machine with an Intel Xeon E5-2650v2 octacore CPU with 32 GB RAM for D^{2}Prox and an origin server. For LTE access, we use SKT as our cellular ISP.

Throughput and CPU Usage: Table III compares the performances of downloading a 250 MB file using D^{2}TP, TCP, UDT (UDP-based reliable data transfer protocol [46]) and IBR-DTN (one of the delay-tolerant network group's disruption-tolerant data transfer implementations [47]) employing TCP as the underlying transport protocol. Both the client and the server run the same transport-layer protocol and a single connection is made between them via an 802.11n Wi-Fi AP. Overall, we find that D^{2}TP's performance is comparable to those of other protocols except for a slight increase in the CPU cycles due to user-level protocol implementation atop UDP. Interestingly, we observe much lower TCP performance on Galaxy S3, which we suspect is due to a kernel implementation bug (after code review) since we do not see the same problem on Nexus 5.

TABLE IV
CONCURRENT CONNECTION PERFORMANCE VIA WI-FI

Protocol (# Connections)	Aggregate Throughput	JFI
TCP (100)	95.1 Mbps	0.99
D^{2}TP (100)	96.3 Mbps	0.99
TCP (50) + D^{2}TP (50)	95.0 Mbps	0.98

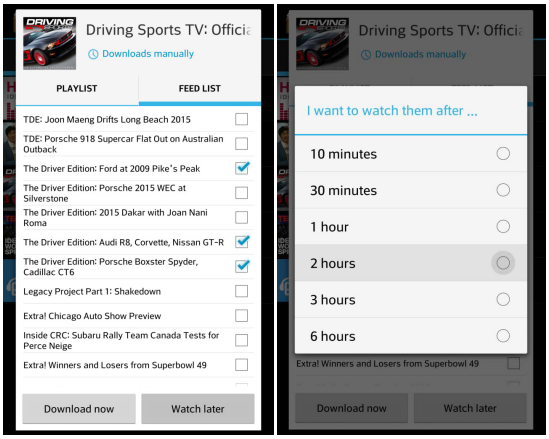


Fig. 5. Screenshot of ReadyCast. Users can specify appropriate deadline for each download.

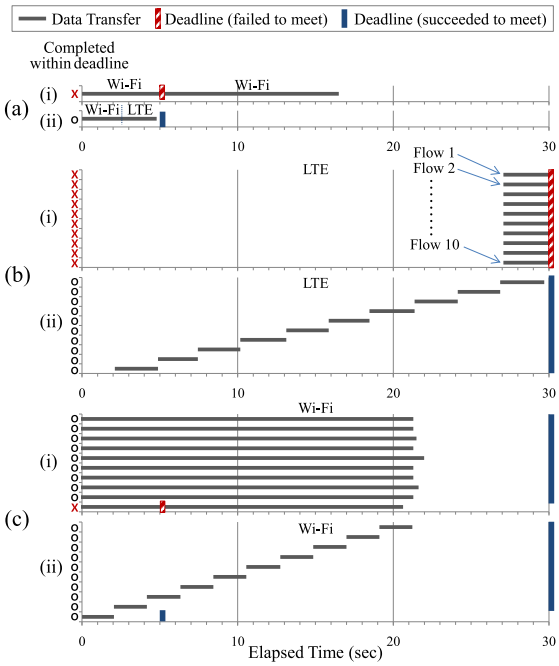


Fig. 6. Timecharts for flow scheduling experiments under (a) poor Wi-Fi connections and (b) and (c) under resource contention by multiple competing flows. Deadlines are shown as vertical bars.

Fairness: We check if D^{2}TP provides a fair bandwidth share among competing flows by measuring Jain's Fairness Index (JFI) [15]. We compare following three cases: *(i)* 100 TCP connections, *(ii)* 100 D^{2}TP connections, *(iii)* 50 TCP and 50 D^{2}TP connections. We have each connection download a 1 GB file on a Nexus 5 device, and show the aggregate throughputs and JFIs in Table IV. We observe that D^{2}TP provides comparable per-flow bandwidth fairness to that of TCP both for D^{2}TP-only flows and for D^{2}TP flows mixed with TCP flows.

Network Switching Delay: We next measure the network switching delays to migrate a D^{2}TP flow between Wi-Fi and cellular networks. We measure the time between the last data packet from the previous network and the first data packet in the new network on a Galaxy S3 device. We measure it for 100 times and show the averages. The switching time includes the source authentication besides hardware interface switching.

We find that it takes 27 to 141 ms to switch from an LTE to a Wi-Fi network with an average of 45 ms. Switching from LTE to Wi-Fi is fast since the device can use LTE until the Wi-Fi access is fully ready. So, the network switching delay is the same as the D^{2}TP source authentication latency. However, in the reverse direction from Wi-Fi to LTE, we see 440 to 856 ms of delay with an average of 581 ms. This is because Android enforces a higher priority level on the Wi-Fi access, which requires the Wi-Fi interface to be turned off before the LTE interface is initialized. We note that it takes 88 ms on average for D^{2}TP source authentication in Wi-Fi-to-LTE switching.

Flow Scheduling With D^{2}Sched: We now evaluate whether D^{2}Sched can guarantee complete data transfer within user's delay tolerance when experiencing poor Wi-Fi connections (Figure 6(a)) or resource contention between multiple flows (Figure 6(b, c)). For these experiments, we use Galaxy S3 and have each flow download a 100 MB file from our server. For each graph, (i) represents a strawman scheduling method (*e.g.,* stick with Wi-Fi if available, schedule each flow independently) while (ii) represents D^{2}Sched's result.

For poor Wi-Fi connection experiment, we run one flow with a short deadline (5 min) in a Wi-Fi network whose bandwidth is lower (0.8 Mbps) than that of a cellular network (4.7 Mbps). This is a valid scenario as it is often the case that a Wi-Fi bandwidth is lower than that of cellular access if the APs are congested or the signal strength is weak [1]. As shown in Figure 6(a)(i), blindly transferring data through Wi-Fi only fails to meet the deadline. On the other hand in (ii), we see that D^{2}Sched automatically switches to LTE to finish the download within flow's deadline. We note that D^{2}Sched still tries to maximize the Wi-Fi offloading ratio, receiving 16% of the data via Wi-Fi, which is close to optimal (17%).

For multiple flow experiment, we first run 10 flows with the same deadline (30 min) in a LTE network without any Wi-Fi availability. The measured bandwidth of the cellular network is 4.7 Mbps on our client. Figure 6(b) compares the behavior of (i) scheduling each flow independently vs. (ii) D^{2}Sched. Under (i), we find that all flows miss their deadlines since they end up waiting for Wi-Fi too long. When they start using LTE, the bandwidth is divided into 10 flows, which delays the flow beyond its independently estimated finish time. In contrast, in (ii), all flows under D^{2}Sched finish in time since it schedules some flows earlier, operating with a minimum required time for meeting the deadlines of all flows. Next, we run 9 flows with a long deadline (30 min) along with one urgent flow with a much shorter deadline (5 min) in a Wi-Fi network.

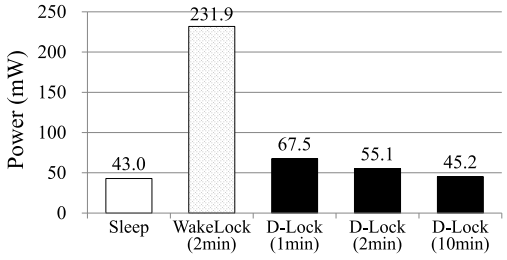


Fig. 7. Idle power usage for WakeLock and D-Lock at various scanning periods.

The measured bandwidth of the network is 6.3 Mbps. Figure 6(c) shows that the urgent flow misses its deadline under (i) since the other 9 flows compete for the bandwidth at the same time even if they have longer deadlines. However, in (ii), D^{2}Sched schedules the urgent flow first while holding off remaining 9 flows, eventually meeting the deadlines of all flows.

Power Usage at Idle States: We show the effectiveness of our D-Lock in reducing power consumption at idle states. We compare the power usage of a Galaxy S3 device when we use *(i)* Android WakeLock (with WifiLock) vs. *(ii)* D-Lock. We set the device to use an LTE network during the experiment, and turn on the Wi-Fi module for periodic scanning. For accurate measurement, we turn off all the other user apps but the built-in apps, which cannot be killed by a user. We use Power Monitor by Monsoon for measurement, and show the results for various scanning periods in Figure 7.

We first see that at least 43 mW is needed to keep the device in a sleep state. If Android WakeLock is acquired, the CPU stays awake for the entire duration of the idle states while the device would scan for a Wi-Fi AP every 2 minute (default scanning policy in Galaxy S3). As a result, the power usage surges up to 231.9 mW, 5.4 times that of sleep state. Given that the battery capacity of Galaxy S3 is 7,980 mWh (3.7V with 2,100 mAh), almost 70% of its total battery would be consumed per day even if no other activity is going on. In contrast, D-Lock with 2 minutes as the scanning period would curb the average power consumption at 55.1 mW, which outperforms WakeLock by a factor of 4.2. This is because the CPU is mostly turned off with D-Lock except when scanning a Wi-Fi AP. Even with a 1-minute period, the power usage reduction is significant (3.4x better than WakeLock), and increasing it to 10 minutes produces the power consumption close to that of sleep states.

Power Usage at Transfer States: We next examine the power usage of D^{2}TP during data transfer in comparison to that of TCP. We measure the power consumption on a Nexus 5 device while downloading a 1 GB file through Wi-Fi using *(i)* TCP vs. *(ii)* D^{2}TP. We find that for 825 seconds of the data transfer (bandwidth = 9.7 Mbps), TCP uses 1043 mW while D^{2}TP spends 1096 mW. This extra 5% of power consumption comes from increased CPU usage from user-level D^{2}TP implementation as is also shown in Table III. However, we believe D^{2}TP will be more power-efficient in the mobility events as it avoids re-downloading the content from the start at network disruptions, also shown in our previous work [16]. Besides, it strives to use power-efficient Wi-Fi as much as possible over power-hungry cellular radio.

D^{2}Prox Scalability: Finally, we evaluate the performance scalability of D^{2}Prox by having it handle a large number of concurrent flows. We employ a client machine of the same specification as the server since it is difficult to create tens of thousands of connections from a single smartphone. We have the client initiate 64K concurrent connections to a D^{2}Prox server, and have each connection download a 1 GB file via a 10 Gbps interface (through a NetGear XSM7224S switch). We make 10% of the active connections go idle to simulate a network disruption for each minute, and when all connections become idle, we make 10% of the idle connections go active again for each minute. We observe that D^{2}Prox achieves almost 10 Gbps throughout the experiment while the bandwidth is evenly distributed among the active flows, producing a JFI above 0.9. Lastly, D^{2}Prox scales linearly as we introduce more D^{2}Prox nodes to the network.

B. Experience With Real-World Applications

We gauge the practicality of Cedos in real-world mobile applications. First, we evaluate if Cedos ensures automatic switching between Wi-Fi and cellular networks even for real-time video streaming with VLC. Second, we carry out a user study with ReadyCast for 8 weeks. We measure Wi-Fi offloading opportunities with delays in real environments, and observe how the users react to delayed data transfers.

1) Opportunistic Data Offloading With VLC: We watch a long streaming video of constant bit rate of 640 Kbps with Cedos-enabled VLC on a Nexus 5 device while riding a subway train in Daejeon, South Korea. The subway line consists of 22 stations, and we take a round trip to go from one end to the other end, and then come back to the starting station. Each subway station is equipped with publicly-accessible Wi-Fi APs but the train itself does not provide Wi-Fi access. So, the Wi-Fi service is available only when the train is staying at a station. Our goal is to opportunistically use the Wi-Fi access to buffer the streaming video but to automatically switch to the cellular network (LTE) if Wi-Fi access is unavailable and the buffered data is too small to be played back. We place D^{2}Prox and an origin server in our lab, which is 2 to 10 Km away from the client with an average RTT of 42.9 ms via LTE and 13.7 ms via Wi-Fi. We set D^{2}BufMgr's *low* and *high* receive socket buffer thresholds to 200 KB and 3 MB, respectively. We set the *low* threshold large enough to cover 0.3 seconds of LTE-to-Wi-Fi interface switching time during video playback. We set the *high* threshold conservatively large since we observe that many public Wi-Fi APs that advertise themselves as accessible actually do not function properly.

Wi-Fi Offloading Ratio: The experiment result shows that as much as 48.7% of the video data is being fetched through public Wi-Fi APs at the stations. For the entire period of video watching, the Wi-Fi data transfer is possible only for 310 seconds with an average of 6.29 Mbps while the device uses LTE for much longer, 2,447 seconds. However, D^{2}BufMgr's buffering mechanism ensures to limit the LTE usage by curbing the content download rate to the video bit

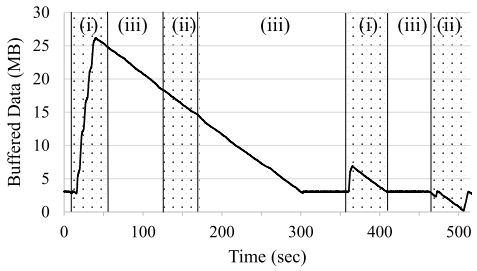


Fig. 8. Buffered data size over time for subway experiment. (i) Wi-Fi data transfer, (ii) Wi-Fi no data transfer, (iii) no Wi-Fi.

rate (640 Kbps). Overall, the result is inspiring since it shows that almost a half of the large streaming data is delivered through Wi-Fi even if Wi-Fi access is available intermittently at a fraction of the stations. We confirm that Cedos effectively offloads the data to Wi-Fi without a single pause in the playback.

Buffer Size at Network Disruptions: We next analyze how a D^{2}TP connection reacts to network disruptions during device mobility. Figure 8 shows the trace of the buffered data size during the time the train passes four subway stations. The shaded portions represent the availability of APs where (i) denotes that the APs are working properly while (ii) marks the period of malfunctioning APs that disallow data transfer. At the 20th second, the client starts buffering the video content through Wi-Fi as much as 26 MB. The device moves out of the Wi-Fi network at the 60th second, but it uses the buffered data for playback until the remaining size hits the *high* threshold at the 300th second. Then, it downloads the video content through LTE, but it keeps the buffer size at the *high* threshold (=3 MB) until it meets a Wi-Fi AP at the next station at the 360th second. The device attempts to buffer the data through the Wi-Fi AP, but the AP seems to be unstable and transfers the data for only about 5 seconds. At the 410th second, the device uses LTE again and when the device meets a malfunctioning AP at the 450th second, it tries to use Wi-Fi only to switch back to LTE at 500th second since the remaining buffer size hits the *low* threshold.

Optimizing Wi-Fi Offloading: We now share our experience in optimizing the Wi-Fi offloading process. In our earlier experiments, we found that only 14.9% of the data was offloaded to Wi-Fi, which appeared lower than expected given the average Wi-Fi attachment time per station is 56.4 seconds while the period of Wi-Fi unavailability between two stations is 60.3 seconds on average. We later found out that once a device is attached to a Wi-Fi AP, it would not rescan for a better AP even if the available bandwidth of the AP is very low or even if it is not working. Actually, Wi-Fi data transfer worked at only 3 out of 44 train stops (round trip of 22-station line) in our earlier experiments.

We fix this problem by rescanning for a better AP every five seconds when a streaming application is running in the foreground. This allows the device to download 3.3 times more data via Wi-Fi at the cost of 8.5% of extra power consumption. With this fix, the number of stations that do transfer the data through Wi-Fi increases to 9 from 3, and the transfer time per station lasts for 34.4 seconds on average. We find that each Wi-Fi transfer at a station downloads 27.1 MB of the data, which is enough for watching the video for more than 5 minutes. Unfortunately, while the other 35 train stops report Wi-Fi availability, we could not transfer any data through their APs. We plan to investigate the reasons in the future.

TABLE V
POWER CONSUMPTION WITH AVERAGE BANDWIDTH OF 1 & 5 Mbps

Wi-Fi	Cellular	Power Consumption	
		1 Mbps	5 Mbps
off	transfer	1,282 mW	1,410 mW
idle	transfer	1,301 mW	1,420 mW
transfer	off	568 mW	686 mW
transfer	idle	576 mW	689 mW

Another problem we found is that it often takes as much as 3 seconds to turn on the 3G cellular interface when we switch from the Wi-Fi interface. Switching to LTE is not a big problem, but if LTE is not available, the device can fall back to the 3G interface. This implies that we need to have a larger *low* buffer threshold to reliably switch from Wi-Fi to cellular interfaces. But having a large *low* buffer threshold could lead to unnecessarily frequent switchings to cellular networks even if Wi-Fi is available (but its available bandwidth is temporarily fluctuating), which would reduce the Wi-Fi offloading opportunities.

One workaround is to keep the cellular interface on even if the device is attached to a Wi-Fi network so that the device could switch to the cellular interface at any time. One concern with the approach is that it would consume more power, which is obviously undesirable. Fortunately, we find that the extra power consumption is reasonable, ranging from only 0.7 to 1.5% as shown in Table V when it is compared with having only the Wi-Fi interface on. So, we decide to keep the cellular interface on even if the device is attached to a Wi-Fi AP.

2) Delayed Data Offloading With ReadyCast: We conduct a user study with ReadyCast by having 50 KAIST students use it for 8 weeks.[2] The students subscribe to any of 3,853 Podcast channels populated in ReadyCast links, and download the contents by setting an allowed delay between 0 to 6 hours. To simulate ISP-embedded D^{2}Prox, we place two D^{2}Prox servers in Korea Advanced Research Network (KOREN) PoPs in Seoul and in Daejeon, respectively. KOREN is a government-funded research network testbed in South Korea, offering high-speed linkage to Japan, China, U.S., and Europe as well. We have ReadyCast choose the nearest D^{2}Prox server by resolving a D^{2}Prox domain name (*e.g.,* `dprox.net`) with our custom DNS server. For each download, the client leaves a per-flow log, which includes a deadline, play timestamps, network disruption/switching records, and data volume transferred via Wi-Fi and cellular connections. Overall, we have collected a total of 2,834 unique connections that complete the downloads, which is 71.2 GB in volume. While the users are mostly staying on campus during weekdays, many of them tend to travel outside the campus during weekends and holidays. So,

[2]We anonymize all usage records for privacy.

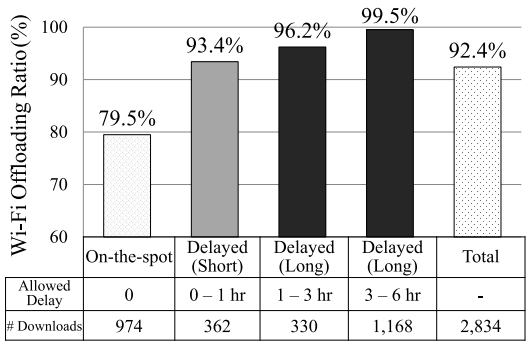


Fig. 9. Offloading ratios by user-configured delays.

the dataset covers remote mobility events beyond the campus networks to some extent.

Network Characteristics: We describe the overall mobile network characteristics logged by ReadyCast. First, we observe that the users are well-exposed to Wi-Fi networks. We find that the user devices are attached to Wi-Fi for 62.9% of the download time. Data transfers via Wi-Fi show a higher average throughput (2.68 Mbps) than that of cellular access (0.82 Mbps), which matches the trend in previous works [2].

336 downloads (11.9%) experience network switchings between Wi-Fi and cellular networks during the download. The average number of network switchings is 2.8 times per download. This means that a network switching is bimodal. The majority of downloads finish within a network, and if there is a network switching (user mobility), the users switch the networks almost 3 times per download before completion. We find that 38% of the entire downloaded data would have been wasted on TCP without application-level resumption, and 80% of the downloads have non-zero user-allowed delays, which implies that the users are willing to benefit from Wi-Fi access even with delays when they expect network switchings while downloading the content.

Wi-Fi Offloading Ratio: We next look at how much data is offloaded to Wi-Fi by ReadyCast. Figure 9 shows the breakdown of Wi-Fi offloading ratios by the delays allowed by the users. Interestingly, even the downloads without any delay ("on-the-spot") deliver 79.5% of the data through Wi-Fi. We suspect that this is because the users choose "no delay" when they already know that they are in the Wi-Fi service area. We find that 70% of the on-the-spot flows download the data completely without cellular access, and only 24% of them receive the content solely from 3G/LTE access. In case a user allows some delays, the offloading ratio increases rapidly. We observe that almost 93% of the data is offloaded to Wi-Fi even for a few tens of minutes of delays. This implies that one does not need to wait very long to meet Wi-Fi APs, and delayed offloading is a practical solution that could significantly reduce the traffic to cellular networks. If the allowed delay exceeds three hours, almost all traffic (99.5%) is offloaded to Wi-Fi. Overall, 92.4% of the total data is delivered through Wi-Fi without any wasted data transfer with ReadyCast.

User Reaction to Delays: We analyze how users react to delayed data transfers by looking at their deadline selection behavior. We first surveyed users' cellular data plans, and found the tendency that the users allow more delays as the monthly data cap is smaller, as shown in Table VI. The actual offloading ratios are in line with the deadline selection where more data is offloaded to Wi-Fi with longer delays.

TABLE VI
USER STUDY STATISTICS BASED ON MONTHLY DATA CAP

User Group with Monthly Limit (x)	Delayed Download	Average Allowed Delay	Offloading Ratio
$0 < x \leq 1$ GB	62.7%	186.1 min	92.6%
1 GB $< x \leq 5$ GB	52.2%	101.7 min	90.7%
$x \geq 5$ GB	47.6%	100.9 min	88.5%

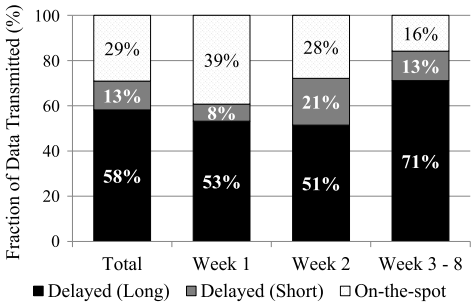


Fig. 10. Delay selection by users (short: $<$ 1hr, long: $\geq$ 1hr).

We next analyze how users' delay selection changes during the experiment period. Figure 10 compares the fractions of user-selected delays for the first two weeks and the remaining six weeks. We see that almost 40% of the bytes are downloaded with no delay for the first week. However, once the users get used to delayed downloads, their delay tolerance grows accordingly, showing as much as 84% of the downloaded volume is tagged with a positive delay and more than 70% of the total bytes allow a delay of one hour or more. This suggests that the users are willing to allow delays as long as the application supports delayed downloads.

User Experience: We conducted a user survey before and after the field study. Before the experiment, almost half of the users responded that they usually keep the Wi-Fi interface off to avoid abrupt connection closures on move. After the experiment, 94% of the users responded that ReadyCast was very useful since it automatically completed the downloads regardless of network disruptions, and 85% have felt almost no increase in battery consumption, implying that D-Lock with the 2-minute polling period works well in practice. Many wanted to extend the service to movies or TV shows, so delay-tolerant Netflix-like service could be promising.

VI. DISCUSSION

In this section, we discuss other issues in developing and deploying delay-tolerant mobile applications.

Leveraging Delay Tolerance: Our work reconfirms that allowing even a slight delay in download offloads a significant fraction of cellular data to Wi-Fi. We believe there are many ways to encourage delayed downloads different from what we have presented in this paper. One simple example is automatic synchronization of periodically-generated contents (e.g., newspapers, TV shows, or even movies) by a daily

TABLE VII
COMPARISON OF THE REQUIREMENTS SATISFIED BY EXISTING NETWORKING STACKS FOR MOBILE APPS AND CEDOS

	TCP migrate	Bundle	Wiffler	Cedos
Handling disruption	✓	✓	✓ (by apps)	✓
Handling long delay	✗	✓	✓ (by apps)	✓
Delayed offloading	N/A	N/A	✓ *	✓
API	BSD	custom	custom	BSD-like

* Note that Wiffler supports delayed offloading for static files only.

deadline. Another way is to develop an app that automatically learns the user's acceptable delay and suggests the user a default value per each download. The app can analyze the content types or the download history and can calculate the time between the download and playback (or usage). Also, if an app can predict the next content to download (e.g., if it can access a playlist in on-line music streaming), it can opportunistically fetch the content through Wi-Fi as we have showed with VLC streaming.

Concurrent Multiple Network Interfaces: In this work, we have not explored using both Wi-Fi and cellular interfaces at the same time, but it could be beneficial if the Wi-Fi bandwidth happens to be too small to finish downloading within a deadline. Allowing to use both networks is in line with adopting multipath TCP in Apple iOS 7 [48] and using a download booster [49] for fast download. To support concurrent multiple network interfaces, we should adjust D^{2}TP to conform to multipath TCP, and fix our flow scheduling algorithm to reflect concurrent data transfers.

Operator-Scale Deployment Cost: We discuss the applicability of D^{2}Prox on a cellular ISP scale. A cellular ISP can decide to deploy D^{2}Prox in a number of vantage points close to the subscribers and allow mobile apps to use D^{2}Prox after authentication. This would efficiently distribute delay-tolerant contents to Wi-Fi with Cedos apps while the remaining interactive, real-time contents are delivered through cellular networks in case Wi-Fi is unavailable. We note the cost of the increasing end-to-end latency due to routing through D^{2}Prox, but we believe most of delay-tolerant flows that benefit from Cedos will be long-lived, large-data transfers that are less affected by it. Moreover, D^{2}Prox could reduce the latency by servicing popular content from its cache without contacting the origin server. By veering a large volume of cellular data traffic to cheap, widely-deployed Wi-Fi APs, Cedos would allow cellular ISPs to reduce the cost of expanding and maintaining the expensive cellular infrastructure.

Handling Hard Failures: We have focused mainly on how to transparently handle soft failures such as network disruptions and delay. However, maintaining delay-tolerant, long-lived flows in the face of hard failures such as app crash or device power outage is another challenge that needs to be addressed. For example, our ReadyCast could resume the data transfer after a hard failure by keeping all flow metadata such as URL, content length, file path and deadline persistent in its flash storage. We plan to support reliable crash recovery in D^{2}TP by using persistently storing flow metadata at minimum overhead [17].

Security: We discuss a few security issues in the operation of Cedos applications. First, D^{2}TP servers (or proxies) could be more vulnerable to state explosion attacks. For example, attackers may issue a large number of D^{2}TP connections with D^{2}Prox, but never download the content from it. Since D^{2}Prox keeps track of many concurrent D^{2}TP sessions, too many idle connections could exhaust the computing resources of D^{2}Prox. Second, attackers may intentionally issue repeated connection or connection resumption requests. Since the connection or connection resumption procedure requires more CPU cycles than regular TCP connection setup, attackers may attempt to use up CPU cycles to deny other requests. However, these attacks are not fundamentally different from other resource exhaustion attacks on regular TCP-based (or TLS-based) servers. One approach to mitigating these problems is to authenticate each app user, and to limit the number of concurrent requests or to enforce a rate on bandwidth consumption per client [18].

VII. RELATED WORKS

There are many related works that address network disruptions or exploit the delay in Wi-Fi offloading. Due to the sheer number, comprehensive treatment is hard here, so we highlight the difference of our approach from other works.

A. Handling Network Disruptions/Delays

There are various ways to handle network disruptions at content download. A popular approach is to resume the download using an application layer protocol (*e.g.,* HTTP range queries). One practical downside, however, is that it requires the modification of each application, which we find is often neglected in mobile apps. Mosh [19] and ATOM [20] provide seamless data transfer at network disruption by synchronizing to the latest state or by querying for the latest download offset. However, their applicability is limited as they base on UDP, or it requires mobile apps to support resumption. 3GPP's I-WLAN [22] specifies a way to integrate the cellular operator's LTE and user-owned Wi-Fi APs. More recently, IFOM [23] was proposed to support seamless flow-level offloading within I-WLAN architecture. However, implementations of I-WLAN and IFOM require tight integration of the two networks, and may result in heavy burden on the backhaul network.

Another workaround is to eliminate the network address binding altogether and to use a unique identifier that persists between network disruptions. Locator/ID Separation Protocol (LISP) [5], i3 [8], and HIP [3] bring in a separate id for the location and use the IP address as an endpoint identifier. However, these protocols require upgrading the existing infrastructure (*e.g.,* routers) or adding a separate entity (*e.g.,* DNS support), which may be challenging for immediate deployment. Mobile IP [24] uses a fixed home address to hide a moving device, which is similar to D^{2}Prox in relaying the

packets. But Cedos supports delayed offloading with a flexible network API that directly controls the mobility events.

TCP Migrate [9] and MPTCP [25], [26] extends TCP to include transport-level flow identifying token. Serval [10] introduces a new service access layer, which demultiplexes packets by flow identifiers and migrates a flow from one address to another. Auspice [27] is a scalable global name service which resolves network location identities under mobility. msocket [28] is a BSD socket-like API which provides mid-connection mobility for mobile-to-mobile and multipath cases by relying on Auspice. However, they all assume a short reconnection and are unable to handle hours of delay as they are based on default TCP teardown after 15 retransmissions [29].

Bundle protocol (BP) [30] is the standard communication protocol for Delay Tolerant Networks (DTN), which enables end-to-end reliable delivery through intermittent connectivity in the challenged networks (e.g., interplanetary [31] or rural Internet access [32]). Notable BP implementations include DTN2 [33], IBR-DTN [34], and ION [35]. Since these protocols are originally designed as transport protocols in multi-hop opportunistic networks, we find that they do not fit well into our system environments as shown in Table VII. First, BP requires bundling the data before sending it, which makes it unsuitable to deliver dynamically generated data (e.g., media streaming). Second, their programming API is largely different from the BSD socket API (e.g., requiring endpoint ID starting with "dtn://"), so that it would be difficult to port the existing mobile applications. Cedos adopts D^{2}TP, a transport layer protocol tailored to Wi-Fi offloading in mobile apps. D^{2}TP extends our early work, DTP [16], a disruption-tolerant transport protocol, by completing the API that allows delay and buffer management with multi-flow scheduling and by supporting D^{2}Prox for easy deployment with realistic mobile apps.

B. Wi-Fi Offloading With Delay

A large number of works have shown the potential of Wi-Fi offloading through mobility studies. A recent study on mobility estimates that over 60% of the cellular traffic can be offloaded to Wi-Fi [1], [2]. Lee et al. [2] report that the offloading ratio would further improve up to 93.7% if the data transfer could be delayed by one hour. User surveys [36], [37], incentive mechanism [38], and economic analysis [39] also support the feasibility of Wi-Fi offloading with delay. These results may change depending on the experimental conditions, but they show that a slight delay helps a lot in increasing Wi-Fi contact chances for most mobile data transfers.

Wiffler [1] implements a Wi-Fi offloading system that leverages user delay tolerance to reduce cellular data traffic. However, Wiffler focuses on exploiting delay only in fixed-sized file transfers, and shifts the burden of handling mobility events to mobile apps. In contrast, Cedos provides transport-layer handling of the mobility events, which allows easy development of mobile apps. Also, since Wiffler does not provide multi-flow scheduling, it could make some flows miss the deadlines when there is resource contention. TUBE [37] builds an app that allows a user to set a delay on mobile apps. However, its application-level implementation requires to jailbreak the device (for autopilot) and to set a deadline per app (not per each flow), which limits flexibility. Like Wiffler, it does not provide transport-layer API nor network disruption handling. In Cedos, we aim to provide flexibility to app developers with minimal migration effort by providing the BSD socket-like API that supports a broad spectrum of apps, ranging from interactive, streaming to non-interactive file transfer apps. SALSA [21] presents an algorithm that trades delay for energy by considering the power cost of a wireless interface and the quality of a wireless channel, whereas Cedos is a system architecture that realizes delayed Wi-Fi offloading.

We note that recent smartphones introduce advanced Wi-Fi network switching algorithms (e.g., "Smart Network Switch" from Samsung Android phones) to improve the QoE by automatically switching the Wi-Fi interface when the signal drops too low or the network is too slow. These features would help D^{2}TP to improve the QoE of the users, but blind switchings agnostic to the need of application workloads (e.g., delay tolerance) could result in increased cellular data usage. In contrast, our solution derives network switching decisions based on the deadline or the amount of data in the buffer to exploit the Wi-Fi network.

VIII. Conclusion

While delay-tolerant Wi-Fi offloading is known to be a great idea, there has been little systems support for it. In this paper, we have presented Cedos, which enables easy development of delay-tolerant mobile apps by hiding the complexity of handling mobility events in a new transport layer, D^{2}TP. D^{2}TP is designed to transparently handle network disruptions and delays, while it schedules multiple flows to meet deadlines for maximal Wi-Fi usage. D^{2}TP also allows real-time buffer management for video streaming through opportunistic Wi-Fi connections. Finally, we provide D^{2}Prox, which enables Cedos apps to coexist with TCP-based servers. We have demonstrated that it is easy to port existing apps to Cedos, and confirmed that the benefit is realized in the real-world usage scenarios. We hope that Cedos further encourages delay-tolerant app development and practicalizes the concept of delay-tolerant Wi-Fi offloading.

Further information on this work can be found on the project's page `http://cedos.kaist.edu/`. The Cedos platform (including D^{2}TP, D^{2}Prox, and ReadyCast) is free software available for download at `https://github.com/dtn-kaist/cedos/`.

References

[1] A. Balasubramanian, R. Mahajan, and A. Venkataramani, "Augmenting mobile 3G using WiFi," in *Proc. ACM MobiSys*, 2010, pp. 209–222.

[2] K. Lee, I. Rhee, J. Lee, S. Chong, and Y. Yi, "Mobile data offloading: How much can WiFi deliver?" in *Proc. ACM CoNEXT*, 2010, Art. no. 26.

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

[4] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, and R. Morris, "Middleboxes no longer considered harmful," in *Proc. USENIX OSDI*, 2004, p. 15.

[5] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis, *The Locator/ID Separation Protocol (LISP)*, document RFC 6830, IETF, 2013.

[6] H. Balakrishnan *et al.*, "A layered naming architecture for the Internet," in *Proc. ACM SIGCOMM*, 2004, pp. 343–352.

[7] A. Feldmann, L. Cittadini, W. Muhlbauer, R. Bush, and O. Maennel, "HAIR: Hierarchical architecture for Internet routing," in *Proc. ACM ReArch*, 2009, pp. 43–48.

[8] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, "Internet indirection infrastructure," in *Proc. ACM SIGCOMM*, 2002, pp. 73–86.

[9] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode, "Migratory TCP: Connection migration for service continuity in the Internet," in *Proc. IEEE ICDCS*, Jul. 2002, pp. 469–470.

[10] E. Nordstrom *et al.*, "Serval: An end-host stack for service-centric networking," in *Proc. USENIX NSDI*, 2012, p. 7.

[11] A. J. Nicholson and B. D. Noble, "BreadCrumbs: Forecasting mobile connectivity," in *Proc. ACM MobiCom*, 2008, pp. 46–57.

[12] O. B. Yetim and M. Martonosi, "Adaptive delay-tolerant scheduling for efficient cellular and WiFi usage," in *Proc. 15th IEEE Int. Symp. A World Wireless, Mobile Multimedia Netw. (WoWMoM)*, Jun. 2014, pp. 1–7.

[13] Y. Kim, J. Lee, J. Jeong, and S. Chong, "Optimal multi-flow scheduling in delayed Wi-Fi offloading," Tech. Rep. 2014. [Online]. Available: http://netsys.kaist.ac.kr/publication/multi-flow.pdf

[14] A. Aguiar and J. Klaue, "Bi-directional WLAN channel measurements in different mobility scenarios," in *Proc. IEEE VTC*, May 2004, pp. 64–68.

[15] R. Jain, A. Durresi, and G. Babic, "Throughput fairness index: An explanation," Ohio State Univ., Columbus, OH, USA, Tech. Rep. ATM Forum/99-0045, 1999.

[16] Y. Go, Y. Moon, G. Nam, and K. Park, "A disruption-tolerant transmission protocol for practical mobile data offloading," in *Proc. ACM MobiOpp*, 2012, pp. 61–68.

[17] Y. Go, N. Agrawal, A. Aranya, and C. Ungureanu, "Reliable, consistent, and efficient data sync for mobile apps," in *Proc. USENIX FAST*, 2015, pp. 359–372.

[18] X. Qie, R. Pang, and L. Peterson, "Defensive programming: Using an annotation toolkit to build DoS-resistant software," in *Proc. USENIX OSDI*, 2002, pp. 45–60.

[19] K. Winstein and H. Balakrishnan, "Mosh: An interactive remote shell for mobile clients," in *Proc. USENIX ATC*, 2012, pp. 177–182.

[20] R. Mahindra, H. Viswanathan, K. Sundaresan, M. Y. Arslan, and S. Rangarajan, "A practical traffic management system for integrated LTE-WiFi networks," in *Proc. ACM MobiCom*, 2014, pp. 189–200.

[21] M.-R. Ra *et al.*, "Energy-delay tradeoffs in smartphone applications," in *Proc. ACM MobiSys*, 2010, pp. 255–270.

[22] *Mobility Between 3GPP Wireless Local Area Network (WLAN) Interworking (I-WLAN) and 3GPP Systems*, Standard 3GPP TS 24.327. [Online]. Available: http://www.3gpp.org/DynaReport/24327.htm

[23] *IP Flow Mobility and Seamless Wireless Local Area Network (WLAN) Offload*, Standard 3GPP TS 23.261. [Online]. Available: http://www.3gpp.org/DynaReport/23261.htm

[24] C. Perkins, *Host IP Mobility Suppore*, document RFC 2002, IETF, 1996.

[25] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar, "Architectural Guidelines for Multipath TCP Development," IETF, RFC 6182, 2011.

[26] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, "Design, implementation and evaluation of congestion control for multipath TCP," in *Proc. USENIX NSDI*, 2011, p. 8.

[27] A. Sharma *et al.*, "A global name service for a highly mobile Internetwork," in *Proc. ACM SIGCOMM*, 2014, pp. 247–258.

[28] A. Yadav, A. Sharma, A. Venkataramani, and E. Cecchet, "msocket: System support for developing seamlessly mobile, multipath, and middlebox-agnostic applications," UMass CS, Amherst, MA, USA, Tech. Rep. UM-CS-2016-010, 2016.

[29] J. Postel, *Transmission Control Protocol*, document RFC 793, IETF, 1981.

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

[31] S. Burleigh *et al.*, "Delay-tolerant networking: An approach to interplanetary Internet," *IEEE Commun. Mag.*, vol. 41, no. 6, pp. 128–136, Jun. 2003.

[32] S. Guo *et al.*, "Very low-cost Internet access using KioskNet," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 5, pp. 95–100, 2007.

[33] M. Demmer *et al.*, "Implementing delay tolerant networking," Intel Research Berkeley, Berkeley, CA, USA, Tech. Rep. IRB-TR-04-020, 2004.

[34] M. Doering, S. Lahde, J. Morgenroth, and L. Wolf, "IBR-DTN: An efficient implementation for embedded systems," in *Proc. ACM CHANTS*, 2008, pp. 117–120.

[35] S. Burleigh, "Interplanetary overlay network: An implementation of the DTN bundle protocol," in *Proc. IEEE Consum. Commun. Netw. Conf. (CCNC)*, vol. 2007, pp. 222–226.

[36] *TUBE Survey*. accessed on Nov. 15, 2016. [Online]. Available: http://scenic.princeton.edu/tube/tdpsurvey.html

[37] S. Ha, S. Sen, C. Joe-Wong, Y. Im, and M. Chiang, "TUBE: Time-dependent pricing for mobile data," in *Proc. ACM SIGCOMM*, 2012, pp. 247–258.

[38] X. Zhuo, W. Gao, G. Cao, and Y. Dai, "Win-Coupon: An incentive framework for 3G traffic offloading," in *Proc. IEEE ICNP*, Oct. 2011, pp. 206–215.

[39] J. Lee, Y. Yi, S. Chong, and Y. Jin, "Economics of WiFi offloading: Trading delay for cellular capacity," in *Proc. IEEE INFOCOM SDP*, Apr. 2013, pp. 3309–3314.

[40] *Android PowerManager WakeLock*. accessed on Nov. 15, 2016. [Online]. Available: https://developer.android.com/reference/android/os/PowerManager.WakeLock.html

[41] *Android Alarm Clock*. accessed on Nov. 15, 2016. [Online]. Available: https://developer.android.com/reference/android/provider/AlarmClock.html

[42] VideoLAN Organization. *Application, VLC Media Player*. accessed on Nov. 15, 2016. [Online]. Available: http://www.videolan.org/vlc/index.html

[43] MozillaWiki. *Application, Mobile/Fennec*. accessed on Nov. 15, 2016. [Online]. Available: https://wiki.mozilla.org/Mobile/Fennec

[44] *Android Application, ReadyCast*. accessed on Nov. 15, 2016. [Online]. Available: https://play.google.com/store/apps/details?id=dtn.readycast

[45] *NetScape Portable Runtime, Mozilla Developer Network*. accessed on Nov. 15, 2016. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSPR

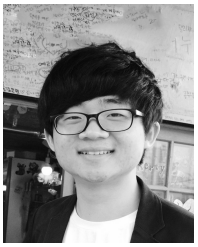
[46] *UDP-Based Data Transfer*. accessed on Nov. 15, 2016. [Online]. Available: http://udt.sourceforge.net/

[47] *IBR-DTN*. accessed on Nov. 15, 2016. [Online]. Available: https://github.com/ibrdtn/ibrdtn/wiki

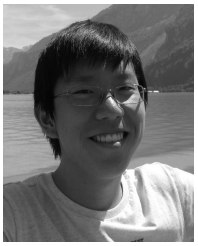
[48] *IOS: Multipath TCP Support in IOS 7*. accessed on Nov. 15, 2016. [Online]. Available: http://support.apple.com/kb/HT5977

[49] *Samsung Galaxy S5 Download Booster*. accessed on Nov. 15, 2016. [Online]. Available: http://galaxys5guide.com/samsung-galaxy-s5-features-explained/galaxy-s5-download-booster/

YoungGyoun Moon received the B.S. degree from the School of Electrical Engineering, KAIST, in 2012. His research interests are in the field of design and implementation of network systems, particularly when it comes to practical issues related to mobile computing, high-performance network data plane, and network function virtualization.

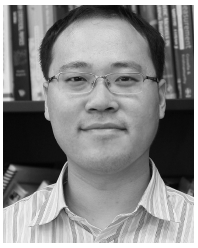


Donghwi Kim received the B.S. and M.S. degrees from the School of Electrical Engineering, KAIST, in 2014 and 2016, respectively. His research interests are in the areas of mobile networking, sensor networking, and network function virtualization.



Younghwan Go received the B.S. degree in electrical engineering and the M.S. degree in electrical engineering and information security from KAIST in 2011 and 2013, respectively. His research interests include designing a new system platform for mobile networks and security, and design and implementation of the next-generation networked and distributed systems.

Yeongjin Kim received the B.S. and M.S. degrees from the School of Electrical Engineering, KAIST, in 2011 and 2013, respectively, where he is currently pursuing the Ph.D. degree. His research interests are in the areas of mobile opportunistic networks, collaborative networking, mobile cloud computing, and electric vehicle charging system management.



Yung Yi received the B.S. and M.S. degrees from the School of Computer Science and Engineering, Seoul National University, South Korea, in 1997 and 1999, respectively, and the Ph.D. degree from the Department of Electrical and Computer Engineering, The University of Texas at Austin, USA, in 2006. From 2006 to 2008, he was a Post-Doctoral Research Associate with the Department of Electrical Engineering, Princeton University. He is currently an Associate Professor with the School of Electrical Engineering, KAIST, South Korea. His current research interests include the design and analysis of computer networking and wireless communication systems, especially congestion control, scheduling, and interference management, with applications in wireless ad hoc networks, broadband access networks, economic aspects of communication networks, and green networking systems. He was a recipient of two best paper awards at the IEEE SECON 2013 and the ACM Mobihoc 2013. He is currently an Associate Editor of the IEEE/ACM TRANSACTIONS ON NETWORKING, the *Journal of Communication Networks*, and *Computer Communications*.

Song Chong (S'93–M'95) received the B.S. and M.S. degrees from Seoul National University, and the Ph.D. degree from the University of Texas at Austin, all in electrical engineering. He was with the Performance Analysis Department, AT&T Bell Laboratories, Holmdel, NJ, USA, as a Member of Technical Staff. Since 2009, he has been the Head of the Computing, Networking, and Security Group, School of Electrical Engineering, KAIST, where he is currently a Professor, and the Founding Director of the KAIST 5G Mobile Communications and Networking Research Center. His current research interests include wireless networks, mobile systems, performance evaluation, distributed algorithms, and data analytics. He received two IEEE William R. Bennett Prize Paper Awards in 2013 and 2016, given to the best original paper published in the IEEE/ACM TRANSACTIONS ON NETWORKING in the previous three calendar years, and the IEEE SECON Best Paper Award in 2013. He is on the editorial boards of the IEEE/ACM TRANSACTIONS ON NETWORKING, the IEEE TRANSACTIONS ON MOBILE COMPUTING, and the IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS. He was the Program Committee Co-Chair of the IEEE SECON 2015 and has served on the Program Committee of a number of leading international conferences, including the IEEE INFOCOM, the ACM MobiCom, the ACM CoNEXT, the ACM MobiHoc, the IEEE ICNP, and the ITC. He serves on the Steering Committee of WiOpt and was the General Chair of WiOpt 2009.

KyoungSoo Park received the B.S. degree from Seoul National University in 1997 and the M.A. and Ph.D. degrees in computer science from Princeton University in 2004 and 2007, respectively. He was an Assistant Professor with the Computer Science Department, University of Pittsburgh, in 2009. He is currently an Associate Professor with the School of Electrical Engineering, KAIST. His research interest is focused on the reliability, performance, and security issues in the design and implementation of networked computing systems. He has developed CoBlitz, a scalable large-file content distribution network, which was acquired by Verivue, Inc., and subsequently by Akamai, Inc. in 2012. He has co-developed HashCache, a memory-efficient caching storage system for developing regions, which was chosen one of the top ten technologies in 2009 by the *MIT Technology Review* magazine. His recent research topics include high-performance packet/flow processing systems using multicore/manycore processors, and novel programming abstractions for complex networked computing systems like stateful middleboxes. He coauthored an mTCP paper that received the community award at USENIX NSDI 2014.