Wireless Network Instabilities in the Wild: Prevalence, App (non)Resilience, and OS Remedy

Zeqi Lai*, Yong Cui*, Yimin Jiang*, Xiaomeng Chen†, Y. Charlie Hu†,
Kun Tan‡, Minglong Dai*, and Kai Zheng‡
* Tsinghua University † Purdue University ‡ Huawei Technologies

***Abstract*— While the bandwidth and latency improvement of both WiFi and cellular data networks in the past decade are plenty evident, the extent of signal strength fluctuation and network disruptions (unexpected switching or disconnections) experienced by mobile users in today's network deployment remains less clear. This paper makes three contributions. First, we conduct the first extensive measurement of network disruptions and signal strength fluctuations (together denoted as instabilities) experienced by 2000 smartphones in the wild. Our results show that network disruptions and signal strength fluctuations remain prevalent as we moved into the 4G era. Second, we study how well popular mobile apps today handle such network instabilities. Our results show that even some of the most popular mobile apps do not implement any disruption-tolerant mechanisms. Third, we present JANUS, an intelligent interface management framework that exploits the multiple interfaces on a handset to transparently handle network disruptions and improve apps' QoE. We have implemented JANUS on Android and our evaluation using a set of popular apps shows that JANUS can (1) transparently and efficiently handle network disruptions, (2) reduce video stalls by 2.9 times and increase 31% of the time of good voice quality compared to naive solutions.**

I. INTRODUCTION

Modern mobile devices such as smartphones and tablets have become ubiquitous thanks to the rapid development of high speed WiFi/cellular technologies. Both WiFi and cellular networks heavily used by mobile handsets today have come a long way, and have seen drastic improvement in bandwidth and latency (e.g. [20], [26]). However, as mobile handsets are intrinsically mobile, to deliver good user experience, merely providing high bandwidth and low latency is not enough, as user mobility inevitably results in wireless signal strength changes and network disconnection and reconnection, collectively denoted as *network instabilities*.

To understand the extent of such network instabilities and hence their impact on the user experience of apps in today's networks, we perform a large scale measurement study of the occurrence of network instabilities experienced by 2000 normal users in their daily life, covering a total of 61112 days and 291 mobile operators (§II). Our study shows that network disruptions remain prevalent in today's WiFi/cellular deployments. In particular, we found (1) On over 30% of the devices, the user experiences more than 25 WiFi disconnections and reconnections and more than 24 cellular data network disconnections and reconnections per day on average. Similarly, on over 10% of the devices, on average the user experiences more than 6.8 and 42 severe signal strength drop episodes of over 10 dBm per day when using WiFi and cellular data, respectively; (2) More importantly, on over 10% of the devices, over 3.9 and 3.0 foreground network sessions in a day experience at least one network disconnection in WiFi and cellular data, respectively. On over 5% of the devices, over 1.7 and 5.3 foreground network sessions in a day experience signal strength decrease of at least 10 dBm, respectively.

Given network instabilities remain commonplace in today's WiFi/cellular deployment, ideally apps are expected to handle network disruptions well. However, our measurement study of a set of popular apps reveals a gloomy picture. In our experiments, we observe even some of the most popular mobile apps do not implement any disruption-tolerant mechanisms (§III). The apps get stuck and fail to automatically resume data transmission when a network disruption happens, which leads to poor user experience. A primary reason for this status-quo is that the lack of programming API support for disruption processing makes it challenging for developers to implement disruption handling for mobile apps [28]. For example, the basic network APIs such as `Socket` and `HttpURLConnection` on Android do not support automatic disruption recovery. These APIs are clumsy at addressing disruptions since they were originally designed for desktop/server environments without network disruption in mind.

Previous efforts have tried to address network disruptions in mobile networks [32], [31]. For example, ATOM [31] offers seamless data transfer upon network disruptions by adding an additional interface switching service (ISS) in the radio access network (RAN). Cedos [32] and MPTCP [41], [23], [16], [27] have been studied to handle network disruptions in the transport layer. However the applicability of these approaches is limited as they require modifying the network infrastructure (e.g. the basestation), the app interface, or the server side. In addition, recent measurements [18], [24], [14], [34], [21] have revealed the inefficiency of MPTCP on mobile devices since MPTCP not only increases the energy consumption by 2.08× on average compared to single path configuration, but also provides very limited throughput improvement (e.g. 1%) or even worse performance for typical apps such as Web, instant message, and VoIP [34].

In this paper, we propose to address the issue by providing direct support inside the OS. Such an approach requires no changes to existing apps nor incurs extra performance degradations and thus is most likely to see wide adoption for

This project is supported by NSFC of China (no. 61422206) and NSF CCF-1320764.

978-1-5090-6501-1/17$31.00

various apps. We first examine the interface selection policies that come with current mobile OSes. Historically, mobile OSes adopted the simple "WiFi-if-available" policy for interface selection [29], for two compelling reasons. First, in the pre-LTE era, WiFi predominantly offers better performance than cellular (2G or 3G) both in terms of bandwidth or latency. Second, WiFi access is typically free (public hotspots or at home) while cellular data usage requires paid subscription.

However, since the wide deployment of LTE/4G, the first reason has largely diminished. In public or outdoor locations – the dominant mobile data access scenarios, WiFi often cannot match the performance of cellular due to uncontrolled contention between public WiFi users, as shown by several recent measurement studies [25], [20], [33]. While the cost reason may still hold, the situation has also improved over the years - the monthly data plan has increased and the cost of data plans has come down as cellular technologies improve (e.g. when it evolved from 3G to 4G). As such, offering "WiFi-if-available" as the only policy today is at the cost of compromised user experience. We argue the OS should at least offer an intelligent interface management framework that exploits multiple interfaces to offer improved user experience by transparently handling disruptions and improving QoE on behalf of unmodified apps. [1]

As the third contribution of the paper, we have designed and implemented such an interface management framework called JANUS (§IV, §V). JANUS is compatible with existing network APIs and does not require modification to the app source code. JANUS has two design goals: (1) *Transparent network disruption handling*: It transparently and correctly handles network level disruptions with little effort from app developers; (2) *Flexibility*: The framework should be flexible to accommodate potentially diverse app network QoE requirements (e.g. real-time or delay-tolerant traffic).

To meet these goals, JANUS integrates three key modules. First, we design *Adaptation Policy*, to allow user to easily specify the app QoE requirements in various scenarios. Second, we design *Link Selector* to properly and continuously select the appropriate interface for each app according to the current interface quality and performance requirement given by the app's adaptation policy. Finally, *Flow Manager* performs transmissions of each flow on the selected interface, while efficiently handling network disruptions.

We have implemented JANUS in Android as a system-level service running in the background that schedules and performs network operations. Our evaluation of the network performance of a set of real world apps shows that on average using JANUS can reduce the number of video stalls of streaming apps by 2.9 times, and increase the duration of good voice quality by 31%. We have open-sourced JANUS at `github`[2] to the Android community to facilitate researchers and developers to improve the user experience of mobile apps.

[1]For users who are constrained by monthly cellular data plans, the OS can extend the intelligent interface framework to further take into consideration the month quota of the data plan. We leave this as future work.

[2]https://github.com/Janus-Anonymous

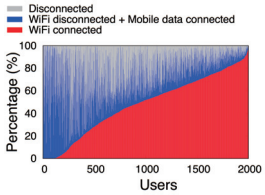


Fig. 1: Distribution of breakdown of time spent in WiFi and cellular states.

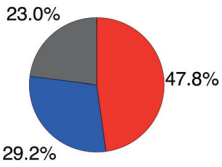
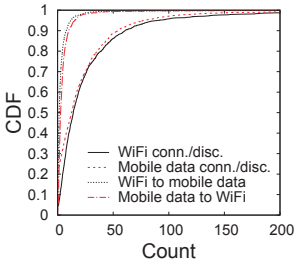
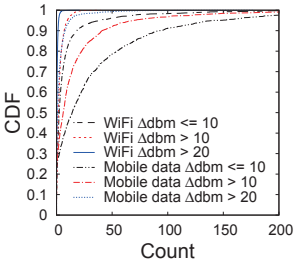


Fig. 2: Average percentage breakdown of time spent in WiFi and cellular states.



(a) Connection and disconnection.



(b) Drop of signal strength.

Fig. 3: CDF of daily average count of WiFi and cellular state changes across users.

II. MOTIVATION

Both WiFi and cellular networks heavily used by mobile handsets today have come a long way, and have seen drastic improvement in terms of bandwidth and latency [26], [20]. Yet little is known about the extent of network disruptions and signal strength fluctuations experienced by mobile handsets in the wild which can directly affect user app experience. In this section, we present a large-scale measurement study of 2000 smartphones in the wild that answers this very question.

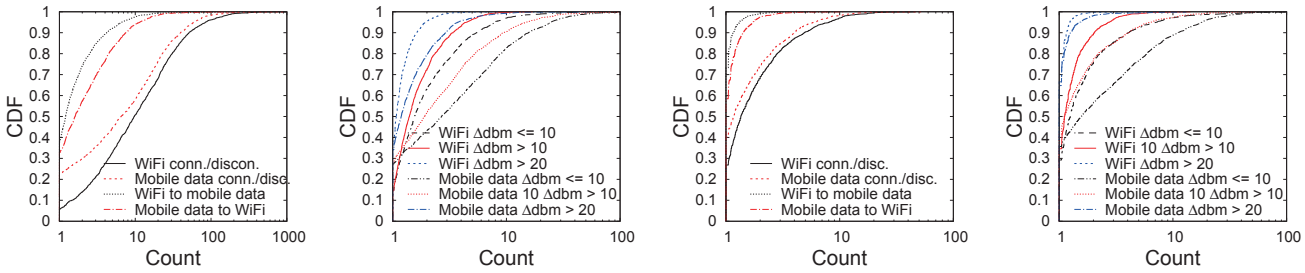
A. Methodology and trace overview

We deployed an utility app called `eStar` [2] in Google Playstore [3]. that has been downloaded on over 100,000 handsets. The app performs periodic logging of the network usage of all apps running on each phone every 5 seconds during screen-off when the CPU is on and every 1 second during screen-on. It also records dynamic events including WiFi connected and disconnected, mobile data connected and disconnected, screen switched on and off, and WiFi/cellular signal strength change. The trace in this study contains logs from 2000 Android devices. The logging of each device last in range of 6 days to 48 days, with an average of 31.9 days. The aggregate trace duration is 61112 days. It covers 342 phone models, 16 Android OS versions and 291 mobile operators.

B. Analysis

Time spent in WiFi and cellular states. Figure 1 shows the distribution of time breakdown spent when (1) WiFi is connected, (2) WiFi is disconnected and mobile data is connected, or (3) neither of them is connected. We see the percentage of time spent when WiFi is connected follows a

[3]Our experiment using the app received exemption from the full requirement of 45 CFR 46 or 21 CFR 56 by the IRB of our organization. In particular, all the data collected from the handsets are anonymized before uploading to the server.



(a) Connection and disconnection. (b) Drop of signal strength. (c) Connection and disconnection during network sessions. (d) Drop of signal strength during network sessions.

Fig. 4: CDF of daily average count of sessions experiencing WiFi or cellular state changes across users.

uniform distribution approximately. Figure 2 plots the average of the 3-way breakdown across the 2000 devices. On average, the devices spent 47.8% of the time connected to WiFi, 29.3% of the time when WiFi is disconnected but mobile data is connected, and the remaining 23.0% of the time when neither of the technologies is connected.

Change of WiFi and cellular states. We next study how often the devices experience state change of network technologies. Figure 3(a) plots the CDF of counts of WiFi and mobile data connection and disconnection, or switching between each other. We observe that 30% of devices experience more than 25 WiFi connections or disconnections per day on average, and similar frequencies for mobile data. The figure also shows 10% of devices experience on average more than 7 times of switching from WiFi to mobile data and more than 10 times of switching from mobile data to WiFi every day. The switching between different network technologies can lead to network disruptions, which affect user experience, resource contention and battery usage. Figure 3(b) plots the CDF of the counts of WiFi and cellular signal strength decrease episodes in three drop levels between the new signal strength and the old signal strength in dBm. We observe that 10% of the devices experience more than 6.8 and 1.3 times of signal strength decrease of more than 10 dBm and 20 dBm for WiFi, respectively, and more than 42 and 7.5 times of signal strength decrease of over 10 dBm and 20 dBm for cellular, respectively. The significant signal strength drop can reduce the network throughput, or even bring disruptions and failure of ongoing network flows [22].

Change of WiFi and cellular states during network sessions. We further analyze the change of WiFi and cellular states when there are active network sessions. Figure 4(a) shows the number of network sessions (i.e. continuous network transmission) experiencing WiFi or mobile data disconnections or switching between the networks. We find that more than 30% of devices have more than 19 and 15 daily network sessions experiencing WiFi and mobile data connection/disconnection, respectively, and more than 10% of the devices have more than 3.2 and 6.1 daily network sessions experiencing switching from WiFi to mobile data and switching from mobile data to WiFi, respectively, which lead to network interruptions and session failure. Figure 4(b) plots the number of network sessions experiencing signal strength decrease of WiFi and cellular. More than 10% of the devices have more than 2.3 and 8.4 daily network sessions experiencing signal strength decreasing of more than 10 dBm in WiFi and cellular, respectively.

We redraw the same statistics in Figure 4(c)(d), but only for network sessions of foreground apps. Because the network traffic of foreground apps is usually directly involved in user interactions, the disruption of the foreground traffic would lead to worse user experience compared to the background traffic. Figure 4(c) shows more 10% of the devices have more than 3.9 and 3.0 daily foreground network sessions experiencing WiFi and mobile data connection/disconnection, respectively, and more than 5% of the devices have more than 0.3 and 0.7 daily foreground network sessions experiencing switching from WiFi to mobile data and from mobile data to WiFi, respectively. Finally, Figure 4(d) shows that more than 5% of the devices have more than 1.7 and 5.3 daily foreground network sessions experiencing signal strength decreasing of more than 10 dBm for WiFi and cellular, respectively.

III. How Well Do Modern Apps handle Network Instabilities?

Given the prevalence of wireless network instabilities experienced by mobile users today, we next examine how existing mobile apps react to dynamic network conditions and network disruptions, by studying the behavior of 30 popular apps selected from Google Play on a Nexus 6 smartphone running Android 6.0.

A. Do apps exploit network interfaces?

We first examine if and how well current apps exploit multiple network interfaces to improve their performance. We examine whether the apps are able to use the best network to obtain good performance under three conditions: (1) *Low WiFi:* We place the smartphone in a location where the WiFi speed is much lower than LTE; (2) *Diminishing WiFi:* We initially connect the smartphone to a high speed WiFi and then walk away from the WiFi AP, and finally we return to the AP; (3) *Congested network load:* We increase the number of running apps and examine how apps load their flows in wireless interfaces.

Table I summarizes the results of how apps react under each condition, where "No support" means that the app does not actively select an interface and passively uses the interface selected by the OS, and "WiFi-only" indicates that the app

TABLE I: Behavior of popular mobile apps upon network disruptions.

Categories	App Name	Network Task	Interface Selection	Recovery Time (s), Disruption Time = 3s, 5s, 10s
Browser	Chrome	Load photo Load dynamic page	No support No support	UI freeze 6.5 / 8.5 / 14.8
	OperaMini	Load static page	No support	UI freeze
Editing	Evernote	Upload notes Move notes	No support	8.4 / No recovery / No recovery No recovery
	OneNote	Upload notes	No support	8.7 / No recovery / No recovery
Local life	Zomato	Load page	No support	No recovery
	Moovit	Route search	No support	No recovery
Trip	Airbnb	Load page	No support	UI freeze
	Momondo			No recovery
	Google Map	Load map	No support	UI freeze
	TouchChina			9.3 / 14.3 / 18.6
News	Flipboard	Load news	No support	UI freeze
	BBC News			No recovery
Shopping	Amazon	Load page	No support	No recovery
	Zappos			
Streaming	YouTube	Streaming Load page	WiFi-only	8.6 / No recovery / No recovery No recovery
	Youku	Streaming Load page	WiFi-only	10.7 / 14.6 / 18.9 No recovery
	JusTalk	VoIP	No support	8.9 / 12.7 / No recovery
	Skype			No recovery
	Douyu Live	Live	WiFi-only	8.7 / 10.3 / 17.6
File download	Dropbox	Upload files Move files	No support	8.8 / 15.5 / 18.7 No recovery
	BaiduCloud	Download file	WiFi-only	9.1 / 12.9 / 17.2
	Google Play	Download file	No support	8.6 / 13.0 / 18.4
Photos	500px	Download photo Upload photo	No support	7.7 / 18.2 / 20.9 No recovery
Social Network	Facebook	Load photo	No support	UI freeze
	Twitter	Load page Load video	No support	No recovery No recovery
	LinkedIn	Load page	No support	No recovery
	Quora	Load page	No support	No recovery

can be optionally configured to only use WiFi. Overall, we find that few apps deploy their own selection mechanisms and the majority of them are forced to use the interface selected by the OS.

However, Android follows a naive policy to use WiFi if available, regardless of the performance of each network. Such a policy is not intelligent enough to sustain good QoE, for either throughput-sensitive or delay-sensitive apps. First, the current policy is *inflexible* to satisfy the performance requirements of various apps. For example, in scenario (1) it is reasonable to load a cost-sensitive task like file downloading on the slow WiFi network, but a real-time app may expect to use the high speed LTE network to achieve better QoE.

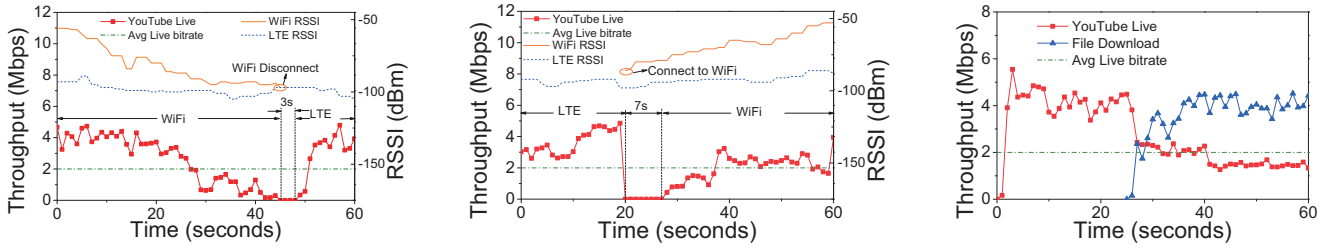
Second, the current policy is unable to adapt to dynamic network conditions. Figure 5a shows an example when we run the YouTube app in scenario (2). As the device leaves the current network, the WiFi signal strength deteriorates and the throughout becomes gradually lower than the required bitrate of the app, but the device remains connected to the WiFi network. Eventually, the connection is lost for several seconds before the device reconnects to the LTE network. Perhaps more interestingly, Figure 5b shows the mirroring behavior happens when the devices moves back to the WiFi network.

Finally, the current policy provides a *coarse-grain* flow management that loads all flows in the same interface. Figure 5c depicts an experiment in scenario (3). Initially we run the Youtube app on the device to play a video. After a few seconds, a background app in the same device starts a download task. The OS loads all connections in the WiFi interface and as a result the throughput of YouTube fails to satisfy the required bitrate due to bandwidth competition. In summary, we found *most apps passively follow the OS WiFi-if-available policy, regardless of the network performance.*

B. How well do current apps react to network disruptions?

Given the prevalence of network instabilities in mobile networks, mobile apps are expected to be able to handle them to ensure good user experience. We next conduct experiments to examine how existing apps react to network disruptions. To emulate a network disruption, we set the device to connect to the WiFi AP first, and turn off both interfaces in the middle of data transmission to incur a network disruption lasting for 3/5/10 seconds before letting it connect to LTE. Table I summarizes the app reaction to such disruptions.

The results reveal a gloomy picture, that in practice most current apps fail to effectively handle network disruptions,



(a) Move from WiFi to LTE. (b) Move back from LTE to WiFi. (c) Bandwidth competition.

Fig. 5: Drawbacks of the WiFi-if-available interface selection policy.

which lead to poor user experience. First, we find that many mobile apps, e.g., OperaMini and BBC News, do not handle network disruptions at all. For example, it is surprising that Opera, a mature browser, fails to reload a web page when we quickly switch the device from a WiFi network to a cellular network. In this case, the page loading gets stuck when the disruption happens. Second, some apps are only able to tolerate very short disruptions but are not robust against longer disruptions. For example, we find that JustTalk fails to resume data loading when it suffers from a network disruption longer than 5 seconds. Finally, interestingly we observe that different tasks performed by the same app may tolerate disruptions differently. For example, the file moving operation in Dropbox cannot resume from any network disruptions while its background file sync operation retries to reestablish connections after a few seconds.

Moreover, although some apps are able to resume connections from network disruptions, their reaction to connectivity recovery of the underlying network is too slow, which also causes poor user experience. When we set the disruption time to 3/5/10 seconds, we observed that the recovery time between the disconnection and the reconnection is much longer than the disruption time. For example, YouTube takes about 8.6 seconds to resume the playback from a 3-second disruption. This indicates that the apps implement some slow recovery schemes to handle disruptions.

In summary, we found *most apps fail to correctly and efficiently handle network disruptions caused by network switch.*

IV. JANUS DESIGN

The main objective of JANUS is to build a framework that intelligently selects a proper wireless interface for apps according is to the app performance requirement and current link condition. As such, the design of JANUS needs to satisfy the following goals.

Intelligent interface selection to meet various app performance requirements. The framework must meet three requirements: (1) Flexibility – it should be able to satisfy the performance requirement of various app types; (2) Adaptive – it should be adaptive to the dynamic network conditions; (3) Fine-grained interface selection – the selection should be performed on a per-app basis instead of per-user or per-device. The flows of different apps should be allowed to use different network interfaces to achieve suitable performance.

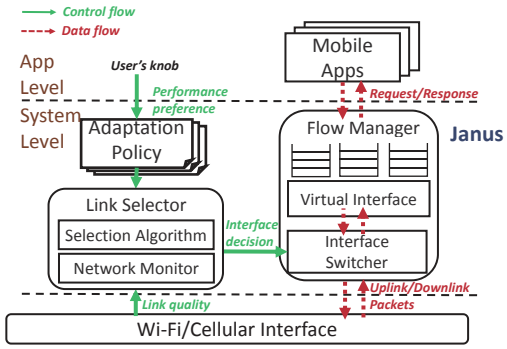


Fig. 6: System Architecture.

Seamless and efficient disruption handling. The framework must seamlessly and efficiently resume from network disruptions with little programming effort from the developers and without too much extra overhead.

Compatibility. The framework must be compatible with the existing APIs. It should be convenient for developers to migrate their existing apps to the framework, and ideally it does not require modifying the app source code, the server side or the network infrastructure (e.g. the base station).

Handling diverse network disruptions. A network disruption can be short or long in duration. We define a *transient network disruption* as an event where a device quickly switches from one network to another and a *permanent network disruption* as an event where the device disconnects from the previous network and there is no new network available for a while. JANUS should transparently handle (1) transient network disruptions for both foreground and background tasks and (2) permanent disruptions for background tasks. This is because if a permanent disruption (e.g. network down) happens when the app is in foreground, it is impossible to quickly resume from the disruption. The proper way to handle this case is to send a failure notification to the user.

A. System architecture

We have designed the JANUS framework following the design goals. Figure 6 shows an overview of the JANUS architecture. At a high level, JANUS runs as a system service that intelligently selects a network interface for apps according to the real-time link quality and the *adaptation policy* configured by the user which describes the performance requirement of the app. JANUS accomplishes this by integrating three key modules, *Adaptation Policy*, *Link Selector* and *Flow Manager*. We discuss them in turn next.

B. Adaptation policies

JANUS provides a knob (implemented in a simple GUI – see §V) to allow a user to customize the performance preference, in the form of an *adaptation policy*, on a per-app basis. The design of adaptation policies follows the following principles: (1) the adaptation policies should cover the performance of various app types, e.g. real-time and delay-tolerant apps; (2) the configuration of adaptation policies should be adaptive to the state of the app. For example an app may require different policies in foreground and background; (3) the adaptation policies should be simple and easy-to-use for users.

Policy types and performance levels. To cover various app types we design three types of adaptation policy: (1) Cost-sensitive policy, which optimizes the data usage in metered networks and is suitable for delay-tolerant apps; (2) Delay-sensitive policy, which optimizes the user-perceived latency and is designed for real-time apps like Web browser and Skype; (3) Throughput-sensitive policy, which needs sufficient bandwidth for video apps like YouTube. Thereafter we denote policy (2) and (3) as Real-time policy. For each policy, we allow the user to manually set the *performance level* which for simplicity is specified as one of several options, high, medium and low, instead of a concrete value.

Common settings. We also consider several common settings for each policy: (1) the priority of the policy. JANUS takes the policy with higher priority into consideration first; (2) data restriction. Since some policies may incur additional cellular data usage to optimize performance, we allow the user to set a restriction on cellular network usage; (3) app state, e.g. background or foreground. A policy can be optionally applied for the background or foreground traffic of the app.

C. Link selector

While an app is running, link selector chooses the best interface for the app according to the specified adaptation policy together with the underlying link quality.

a) Interface selection for real-time Apps: For delay-tolerant policies, JANUS schedules app flows to be clustered and only transferred via WiFi network. To perform selection for Real-time policies, our basic idea is to load the flow to an interface that best fits the performance (throughput/RTT) level. In particular, the selection is expected to satisfy the following principles: (1) to save mobile traffic, the algorithm should prefer to use WiFi interface if both interfaces satisfy the performance level; (2) the selection process should not incur high latency during a switch process; (3) the selection algorithm should not be too sensitive to a temporary poor link quality because interface switching may interrupt the ongoing connection. For example, the WiFi interface may suffer from poor quality for a short time if the device temporarily moves to a place with poor signal strength. In this scenario, it is undesirable to terminate and switch an ongoing flow.

To meet the first principle, we divide the WiFi-LTE performance requirement (PR) space into three regions or cases as shown in Figure 7. In Case 1 where the WiFi performance is better than the required value, we select WiFi for apps. In

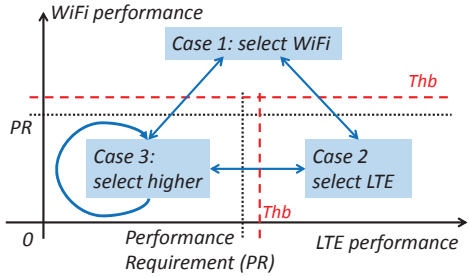


Fig. 7: Interface selection in various network conditions. Case 1: WiFi performance is higher than PR; Case 2: only LTE performance outperforms PR; Case 3: both WiFi and LTE cannot fit PR.

Case 2, the LTE performance is higher than the requirement while the WiFi performance is lower. In this case we select LTE. In Case 3 where both WiFi and LTE performance cannot meet the required value, we prefer to use the interface with higher performance to improve the user experience.

We proposed a threshold-based control to satisfy the second principle. In practice, if we turn on the backup interface when we need to switch a flow, the switching process inevitably suffers 1-3 seconds hardware latency caused by turning on the interface. One approach to avoid this latency is to keep both interfaces on, but such a method may incur much energy drain. We design a performance threshold that is a slightly larger than the performance requirement as Thb = (1+β)*$required\_performance$. To reduce the switch latency, during a switching process we turn on the backup interface if the performance of current interface is below Thb to prepare for the switch.

To follow the third principle, we propose a time threshold Tht to avoid the extra overhead caused by frequent interface switches. For the current interface in use, if its throughput is below Thb and cannot satisfy the performance requirement, we do not immediately switch to the backup interface. If the current interface still cannot satisfy the performance after Tht seconds, and the backup interface is available, we switch the flow to the backup interface to guarantee good user experience.

Table II describes the conditions in which we switch flows from the current interface to the backup interface in each case. Thb and Tht are the two thresholds used to reduce switch latency and control the switch frequency. T is the time when the throughput of current interface is below Thb. T_W and T_L are the performance of WiFi and LTE respectively. For example, if the device is currently in case 1 and using the WiFi interface, JANUS then decides to use the LTE interface if the WiFi performance decreases below Thb for more than Tht seconds while $T_L > required\_performance$.

b) Monitoring network state and performance: We design a network monitor module to monitor the state and performances of underlying networks. In practice, calling Android libraries to extract the network state takes a few seconds to detect the network state change as shown in Figure 5. To reduce the monitoring delay, we leverage Netlink [5] to design a native listener to bypass the framework and watch

TABLE II: Details of the interface selection: conditions that trigger the switch from the current interface to the backup interface. Thb: performance threshold; Tht: time threshold; T_W: WiFi performance; T_L: LTE performance.

Current Case	Switch Conditions
Case 1 (WiFi)	$T_W < Thb$, $T > Tht$, $T_L > PR$ (to C2), $PR > T_L > T_W$, $T > Tht$ (to C3.1)
Case 2 (LTE)	$T_W > Thb$, $T > Tht$ (to C1) , $PR > T_W > T_L$, $T > Tht$ (to C3.2)
Case 3.1 (LTE)	$T_W > Thb$, $T > Tht$ (to C1) , $PR > T_W > T_L$, $T > Tht$ (to C3.2)
Case 3.2 (WiFi)	$T_L > Thb$, $T > Tht$, $T_W < PR$ (to C2), $PR > T_L > T_W$, $T > Tht$ (to C3.1)

out for the network state change in the kernel space.

We monitor the cellular performance following the methods in [38], [37], [17] which leverage signal strength to estimate the cellular performance. Since WiFi is free and more likely to be congested, the monitor module actively probes the WiFi throughput every 30 seconds to monitor the WiFi performance.

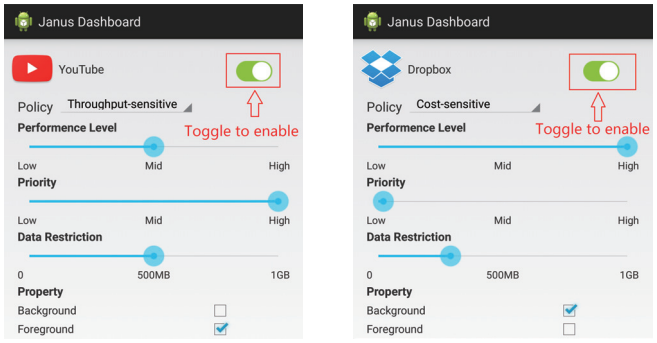
D. Flow manager

JANUS's flow manager is responsible for (1) loading network traffic in a selected interface on a per-app basis; (2) transparently and efficiently handling network disruptions.

Masking network disruptions for apps. Typically, apps manipulate network connections via network APIs such as `Socket` and `HttpURLConnection`. When a network disruption occurs, these APIs throw an exception to the developers and expect apps to handle disruption by themselves. To make the disruption recovery transparent and compatible with the apps, we design a virtual interface between apps and the underlying network interfaces. In particular, we tunnel all app traffic using iptables and Sock5 proxying [9] which changes the destination IP and port to redirect app flows to the virtual interface. The Socks5 protocol only adds a very small header to each packet and its impact on the traffic pattern is negligible [34]. In this way, apps are transparently connected to the virtual interface, and thus network disruptions from physical interfaces are not thrown to apps directly. Instead, recovery operations are performed under the virtual interface.

Recovery for network disruptions. JANUS's flow manager includes an interface switcher component to switch flows according to the decision made by link selector. To handle a transient network disruption, JANUS quickly retries to connect to the server after the termination of current connections. If it is still unable to connect to the server after retrying for several times, the device may suffer a permanent disruption. To handle a permanent disruption (e.g. network down), JANUS stores the states of ongoing flows and releases WiFi/CPU wakelocks to avoid extra energy drain. We extend the AlarmClock class in Android libraries to set a hardware timer to wake up the device and resume transmission if the network is available again.

To reduce the traffic overhead during the switch, JANUS's interface switcher leverages three observations: (1) HTTP is the dominant mobile protocol, and HTTP-based video traffic accounts for more than 75% of the total bytes carried on mobile networks [1]; (2) most web servers today have adopted HTTP progressive download [6] by default. Through this scheme, JANUS's flow manager requests data in byte ranges instead of re-transferring data from the start when resuming an interrupted connection; (3) for most HTTP-based streaming and browsing apps, the content within a session is downloaded using multiple HTTP-GET requests over time [31]. Therefore, when JANUS switches the interface, flow manager simply performs subsequent HTTP-GET requests over the new interface.



(a) YouTube. (b) Dropbox

Fig. 8: JANUS GUI: configuring adaptation policies for YouTube and Dropbox.

V. IMPLEMENTATION

We have implemented JANUS framework as a system level service on LG Nexus 6 running Android 6.0 with a 2.0 GHz octa-core 64-bit CPU and 3GB RAM. JANUS is implemented in around 2700 lines of Java code. In our deployment, we map the options in an adaptation policy to concrete values. Specifically in the Throughput-sensitive policy, we map the high/medium/low performance levels to 1/3/5 Mbps throughput, respectively. We choose these values because they are the recommended bitrate for playing a low/standard/high definition video [8] in typical mobile streaming apps (e.g. Youtube). In the Delay-sensitive policy, we map the high/medium/low levels to 200/400/600 ms latency, respectively following the performance level of Delay-sensitive apps suggested by [11]. In the Cost-sensitive policy, we interpret the high level as only transferring data via WiFi to optimize cellular usage, and medium/low levels as deferring transfers to WiFi but within the deadline of 1 day/hour. We set the bandwidth threshold $\beta = 0.2$ and timer $Tht = 3s$ in our current implementation.

Figure 8 shows JANUS's GUI for letting the user configure adaptation policies for individual apps. In the examples, the user set a Throughput-sensitive policy with the medium performance level and high priority for the foreground traffic of YouTube, and a Cost-sensitive policy for Dropbox to synchronize its background traffic if WiFi is available.

VI. EVALUATION

A. Microbenchmark

We first use a microbenchmark to evaluate the basic performance of JANUS as an integrated system. We leverage

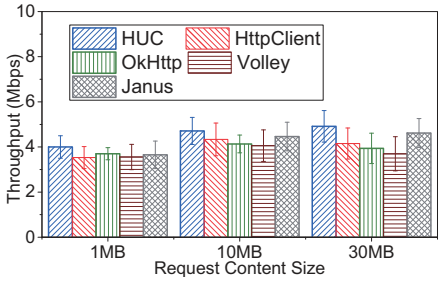


Fig. 9: Throughput comparison.

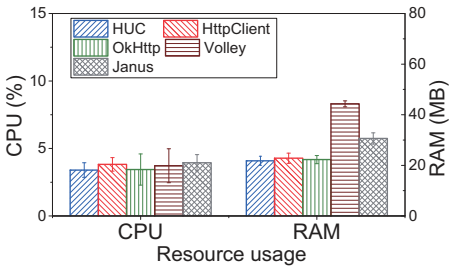


Fig. 10: CPU and RAM.

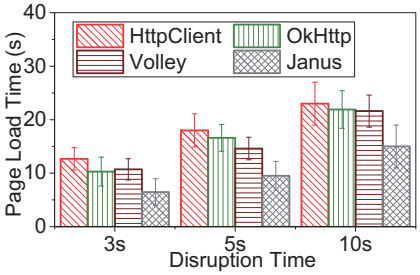


Fig. 11: Load time.

HttpURLConnection (HUC) which is one of the basic network API on Android to implement a toy app that performs a simple function to send HTTP requests and load a web page. We run the toy app with and without JANUS. For comparison, we implement the same functionality using three API-level approaches, Volley [10], OkHttp [7] and Apache HttpClient [3], which are three popular network libraries on Android.

Throughput, CPU and RAM usage. We compare the throughput, CPU and RAM usage of different implementations in loading pages of different sizes. For each implementation, we run each test 5 times and calculate the average result. The results are shown in Figure 9 and 11. As the basic network API HUC gains the highest throughput and the lowest CPU/RAM overhead for different request sizes. However, JANUS's throughput is higher than that of the other three libraries and is very close to that of HUC. The average CPU and RAM usage of each approach are similar except Volley. Interestingly, we find that Volley incurs much more RAM overhead than other implementations which we suspect is due to problems in its internal implementation.

Effectiveness on handling network disruptions. Next, we examine the ability and efficiency of handling network disruptions. We follow the same methodology in Section 3 to inject a network disruption in the middle of data transmission. Since HUC is the basic API without disruption handling function, we only evaluate the network disruption processing of JANUS and other three approaches. As shown in Figure 11, we find that JANUS reduces the page load time (including both data transfers and disruption recovery) by up to 37.18/34.93/30.56% for a 3/5/10s transient network disruption as compared to other approaches. We find these three libraries periodically check the connectivity to resume the connection, and the recovery time significantly depends on the checking interval. The benefit of JANUS comes from asynchronously monitoring connectivity and quickly reacting to underlying network state.

B. Experience with real-world apps

Handling network disruptions. We replay the experiment conducted in Section III to execute specific network tasks in real apps and examine how JANUS helps to effectively handle network disruptions. We enable JANUS for apps in Table I in the dashboard as shown in Figure 8 and set a proper policy for each app according to their types. First, we find that with JANUS, the apps in Table I that do not handle network disruptions (e.g. BBC News) are tolerant to network disruptions. Transient disruptions and background permanent disruptions are transparently handled by JANUS and do not interrupt user interactions. Foreground permanent disruptions are left to apps since they should be reported to users.

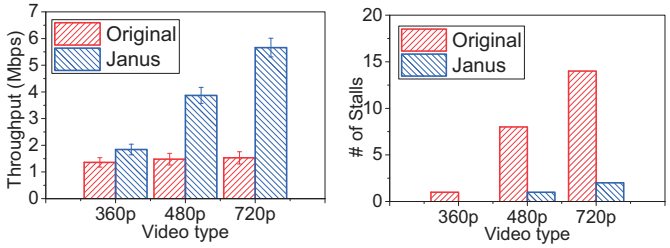
Second, for the apps that are able to resume from disruptions themselves as shown in Table I, JANUS reduces their recovery time. To evaluate the efficiency improvement on handling network disruptions, we plot their disruption recovery time (including data transfers and disruption time) with and without JANUS. As shown in Table III, JANUS reduces the recovery time of Google Play Store, 500px and TouchChina by up to 38.75%, 51.50%, and 40.05%, respectively. Results of other apps are similar and omitted due to the page limit. The improvements come from that JANUS monitors the recovery of underlying networks and immediately resumes connections once the network is available again.

Flexibility. To demonstrate the flexibility of JANUS, we run YouTube to play videos in different bitrate under a low WiFi. Figure 12a shows the throughput of playing video at different resolutions. As the required bitrate increases, JANUS loads the streaming flow in a proper interface that meets the performance requirement, while the original mechanism fails to flexibly select a right interface to sustain good QoE. Figure 12b plots the number of stalls during the playback process. JANUS reduces the number of stalls by 85.7% as it switches to LTE when the WiFi throughput is insufficient.

Adaptation to dynamic network conditions. We next evaluate how JANUS adapts to the dynamic network conditions. We run a VoIP app (JusTalk) and a live streaming app (YouTube) on the device, and walk from a WiFi network to a LTE network. As shown in Figure 13, the original policy only switches to use LTE when the WiFi connectivity is totally lost. As a result, the voice call is interrupted for about 6 seconds. In contrast, JANUS quickly switches to the better

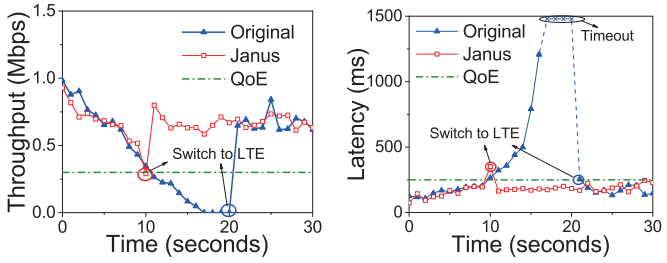
TABLE III: The recovery time of real-world apps under disruption time 3s/5s/10s.

App Name	Network Task	Recovery Time (s) [JANUS]{Default}
Google Play	Download APK Files	[4.4/6.5/13.6] {8.6/13.0/18.4}
500px	Load Image Page	[3.9/6.1/12.7] {7.7/18.2/20.9}
TouchChina	Load News	[4.9/7.0/13.4] {9.3/14.3/18.6}



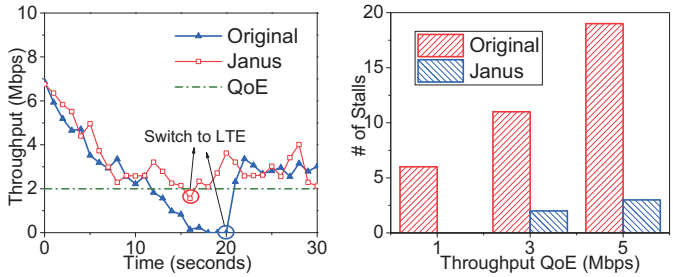
(a) Achieved bitrate of videos. (b) Number of stalls.

Fig. 12: JANUS meets the performance requirement of apps.



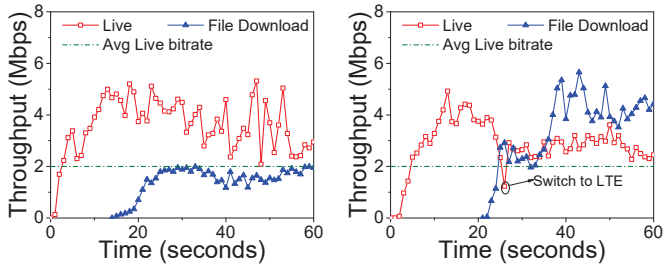
(a) VoIP. (b) Latency over time.

Fig. 13: JANUS properly selects wireless interface to reduce latency of the VoIP app.



(a) Live streaming. (b) Number of stalls.

Fig. 14: JANUS selects the best interface to sustain good user experience for live streaming.



(a) Cellular not allowed. (b) Cellular allowed.

Fig. 15: Flows of different apps are assigned to different interfaces to sustain good QoE.

interface when the WiFi performance decreases and sustains good performance at mobility.

Similarly, because the original policy statically selects WiFi, the bitrate requirement of the streaming apps cannot be satisfied when the WiFi throughput decreases as we leave the WiFi AP, as shown in Figure 14. In contrast, JANUS dynamically makes interface decision and quickly switches to LTE when the WiFi throughput is low, and sustains smooth playback for the streaming app.

Fine-grained flow control. We then run YouTube for live streaming, together with a file download app on the same device. The download app starts a download task in the middle of playing video. First, we set the live app to have higher priority than the download app, and only allow loading streaming flows in the WiFi interface. The app throughput are shown in Figure 15a. In this case, JANUS limits the download throughput to sustain the performance requirement for the live app. Then we set the download app to have higher priority than the live app and allow the live app to use LTE interface. Figure 15b shows when the download task starts, JANUS switches the streaming flow to LTE to avoid bandwidth competition. Summarily, JANUS's fine-grained flow management is able to intelligently load flows in different interfaces to achieve good QoE.

C. Experiments in outdoor environments

We finally evaluate the effectiveness of JANUS in outdoor environments. We set the Delay-sensitive policy for a VoIP app and the Throughput-sensitive policy for a streaming app. We run the apps in outdoor environments while recording the throughput/RTT in every second to generate 3194 VoIP runs and 7210 streaming samples. We replay the traces under the original WiFi-if-available policy and JANUS. Table IV shows the results. For the VoIP app we use mean-opinion-score (MOS) [4] to evaluate the quality of voice talk and consider the quality is good if MOS is larger than 3 [13]. Since JANUS is adaptive to the dynamic network condition at mobility and quickly switches to the best interface during a voice session, it reduces the average end-to-end latency by 33% while increasing the time with good quality by 31%. For the streaming app, each sample lasts for about 5 minutes. JANUS improves the user experience by reducing the average number of stalls per sample by 2.9X and the rebuffering time by 3.3X.

VII. RELATED WORK

We discuss closely related work in several areas.

Characterizing WiFi/cellular networks. Several prior works measure the performance of WiFi and cellular networks. Balasubramanian et al. [12] measured the access and performance of WiFi and 3G networks from moving vehicles in different cities. The authors of [20], [18], [24] measured and compared WiFi and cellular network performance on the same device at the same time. Ding et al. [22] performed the first measurement study of 3G and WiFi signal strength experienced by a large number of smartphone users and quantified the energy impact of poor signal strength on data transfers. In contrast, our measurement study focuses on the instabilities of WiFi/cellular networks experienced by the handsets.

Handling network disruptions. A few studies tried to handle network disruptions from the network side. We already discussed infrastructure approaches such as ATOM [31] and library approaches (§I) and their limitations. Several studies such as TCP Migrate [40], MPTCP [23], [34], [41], Cedos [32], proposed to extend the TCP/IP stack to handle network disrup-

TABLE IV: JANUS improves the user experience for VoIP and streaming apps in outdoor environments.

	JusTalk (VoIP)			YouTube (Streaming)		
	# of Runs	% time voice quality good	Average Latency	# of Sample	Average rebuffering time (s) per sample	Average # of stall per sample
Original	3194 (11760 mins in total)	51%	305 ms	7210	26.47	11.6
JANUS		82%	208 ms		8.09	4.0

tions. However, such approaches require modifying the TCP/IP stack and none of them provides the flexibility to express diverse app requirement. In contrast, JANUS is compatible with existing protocols (e.g. TCP and HTTP) while providing a knob for configuring various performance requirements.

Interface selection. Several studies [19], [15], [21], [36], [42] proposed algorithms for interface selection or enabling co-existence between WiFi and LTE, but only solved a part of the problem since they did not provide a practical solution to handle network disruptions. A few studies [12], [29] focused on scheduling user data across WiFi and cellular interfaces, but were limited to delay-tolerant traffic and did not provide support for real-time apps. In addition, there are other systems complementing our work by studying on energy-efficient interface selection [35] and improving WiFi utilization [39], [30].

VIII. CONCLUSION

In this paper, we propose JANUS, an intelligent interface management framework in the OS that exploits the multiple interfaces to transparently handle network disruptions and meet apps' performance requirement. Our extensive evaluation using real-world apps showed that JANUS can transparently and efficiently handle disruptions and improve performance.

REFERENCES

[1] Ericsson traffic report. http://tinyurl.com/l4sg5td.
[2] Estar. https://play.google.com/store/apps/details?id=com.mobileenerlytics.estar&hl=en.
[3] Httpclient. https://hc.apache.org/httpcomponents-client-ga/.
[4] Mos. https://en.wikipedia.org/wiki/Mean_opinion_score.
[5] Netlink. http://man7.org/linux/man-pages/man7/netlink.7.html.
[6] Nginx. https://www.nginx.com/blog/smart-efficient-byte-range-caching-nginx/.
[7] Okhttp. http://square.github.io/okhttp/.
[8] Recommended video bitrate. https://support.google.com/youtube/answer/1722171?hl=en.
[9] redsocks. http://darkk.net.ru/redsocks/.
[10] Volley. http://developer.android.com/training/volley/index.html.
[11] A. Arjona, C. Westphal, A. Ylä-Jääski, and M. Kristensson. Towards high quality voip in 3g networks-an empirical study. In *AICT*, 2008.
[12] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting mobile 3G using WiFi. In *MobiSys*, 2010.
[13] A. Balasubramanian, R. Mahajan, A. Venkataramani, B. N. Levine, and J. Zahorjan. Interactive wifi connectivity for moving vehicles. *ACM SIGCOMM Computer Communication Review*, 2008.
[14] Y.-C. Chen, Y.-s. Lim, R. J. Gibbens, E. M. Nahum, R. Khalili, and D. Towsley. A measurement-based study of multipath tcp performance over wireless networks. In *IMC*. ACM, 2013.
[15] P. Coucheney, C. Touati, and B. Gaujal. Fair and efficient user-network association algorithm for multi-technology wireless networks. In *INFOCOM*. IEEE, 2009.
[16] A. Croitoru, D. Niculescu, and C. Raiciu. Towards wifi mobility without fast handover. In *NSDI*, 2015.
[17] Y. Cui, S. Xiao, X. Wang, M. Li, H. Wang, and Z. Lai. Performance-aware energy optimization on mobile devices in cellular network. In *IEEE INFOCOM*, 2014.
[18] Q. De Coninck, M. Baerts, B. Hesmans, and O. Bonaventure. A first analysis of multipath tcp on smartphones. In *PAM*. Springer, 2016.
[19] S. Deb, K. Nagaraj, and V. Srinivasan. Mota: engineering an operator agnostic mobile service. In *MobiCom*. ACM, 2011.
[20] S. Deng, R. Netravali, A. Sivaraman, and H. Balakrishnan. Wifi, lte, or both?: Measuring multi-homed wireless internet performance. In *IMC*. ACM, 2014.
[21] S. Deng, A. Sivaraman, and H. Balakrishnan. All your network are belong to us: A transport framework for mobile network selection. In *HotMobile*. ACM, 2014.
[22] N. Ding, D. Wagner, X. Chen, A. Pathak, Y. C. Hu, and A. Rice. Characterizing and modeling the impact of wireless signal strength on smartphone battery drain. In *SIGMETRICS*, 2013.
[23] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural guidelines for multipath tcp development. Technical report, IETF RFC 6182, March 2011.
[24] B. Han, F. Qian, S. Hao, and L. Ji. An anatomy of mobile web performance over multipath tcp. In *CONEXT*. ACM, 2015.
[25] J. HUANG, F. QIAN, A. GERBER, Z. M. MAO, S. SEN, and O. SPATSCHECK. A close examination of performance and power characteristics of 4g lte networks. In *ACM MobiSys*, 2012.
[26] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck. An in-depth study of lte: Effect of network protocol and application behavior on performance. In *SIGCOMM*. ACM, 2013.
[27] M. Jadin, G. Tihon, O. Pereira, and O. Bonaventure. Securing multipath tcp: Design and implementation. In *IEEE INFOCOM*, 2017.
[28] X. Jin, P. Huang, T. Xu, and Y. Zhou. Nchecker: saving mobile app developers from network disruptions. In *EuroSys*. ACM, 2016.
[29] K. Lee, J. Lee, Y. Yi, I. Rhee, and S. Chong. Mobile data offloading: how much can wifi deliver? In *CONEXT*. ACM, 2010.
[30] C.-H. Lin, Y.-T. Chen, K. C.-J. Lin, and W.-T. Chen. acpad: Enhancing channel utilization for 802.11 ac using packet padding. In *IEEE INFOCOM*, 2017.
[31] R. Mahindra, H. Viswanathan, K. Sundaresan, M. Y. Arslan, and S. Rangarajan. A practical traffic management system for integrated lte-wifi networks. In *MobiCom*. ACM, 2014.
[32] Y. Moon, D. Kim, Y. Go, Y. Kim, Y. Yi, S. Chong, and K. Park. Practicalizing delay-tolerant mobile apps with cedos. In *MobiSys*, 2015.
[33] A. Nika, Y. Zhu, N. Ding, A. Jindal, Y. C. Hu, X. Zhou, B. Y. Zhao, and H. Zheng. Energy and performance of smartphone radio bundling in outdoor environments. In *WWW*. ACM, 2015.
[34] A. Nikravesh, Y. Guo, F. Qian, Z. M. Mao, and S. Sen. An in-depth understanding of multipath tcp on mobile devices: measurement and system design. In *WWW*. ACM, 2016.
[35] T. Pering, Y. Agarwal, R. Gupta, and R. Want. Coolspots: reducing the power consumption of wireless mobile devices with multiple radio interfaces. In *MobiSys*. ACM, 2006.
[36] A. Rahmati, C. Shepard, C. C. Tossell, L. Zhong, P. Kortum, A. Nicoara, and J. Singh. Seamless tcp migration on smartphones without network support. *IEEE Transactions on Mobile Computing*, 2014.
[37] A. Rahmati and L. Zhong. Context-for-wireless: context-sensitive energy-efficient wireless data transfer. In *MobiSys*, 2007.
[38] A. Schulman, V. Navda, R. Ramjee, N. Spring, P. Deshpande, C. Grunewald, K. Jain, and V. N. Padmanabhan. Bartendr: a practical approach to energy-aware cellular data scheduling. In *ACM MobiSys*, 2010.
[39] H. Soroush, P. Gilbert, N. Banerjee, B. N. Levine, M. Corner, and L. Cox. Concurrent wi-fi for mobile users: analysis and measurements. In *CONEXT*. ACM, 2011.
[40] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory tcp: Connection migration for service continuity in the internet. In *Distributed Computing Systems*. IEEE, 2002.
[41] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *NSDI*, 2011.
[42] S. Yun and L. Qiu. Supporting wifi and lte co-existence. In *INFOCOM*. IEEE, 2015.