Understanding the Networking Performance of Wear OS

XIAO ZHU, University of Michigan, USA
YIHUA ETHAN GUO, Uber Technologies, Inc., USA
ASHKAN NIKRAVESH, University of Michigan, USA
FENG QIAN, University of Minnesota – Twin Cities, USA
Z. MORLEY MAO, University of Michigan, USA

Networking on wearable devices such as smartwatches is becoming increasingly important as fueled by new hardware, OS support, and applications. In this paper, we conduct a first in-depth investigation of the networking performance of Wear OS, one of the most popular OSes for wearables. Through carefully designed controlled experiments conducted in a cross-device, cross-protocol, and cross-layer manner, we identify serious performance issues of Wear OS regarding key aspects that distinguish wearable networking from smartphone networking: Bluetooth (BT) performance, smartphone proxying, network interface selection, and BT-WiFi handover. We pinpoint their root causes and quantify their impacts on network performance and application QoE. We further propose practical suggestions to improve wearable networking performance.

CCS Concepts: • **Networks → Network protocols**; **Network performance evaluation**; • **Human-centered computing → Ubiquitous and mobile devices**;

Additional Key Words and Phrases: Wearable; Bluetooth; WiFi; Proxy; Interface Selection; Handover

ACM Reference Format:
Xiao Zhu, Yihua Ethan Guo, Ashkan Nikravesh, Feng Qian, and Z. Morley Mao. 2019. Understanding the Networking Performance of Wear OS. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 1, Article 3 (March 2019), 25 pages.
https://doi.org/10.1145/3311074

1 INTRODUCTION

Smart wearable devices are becoming increasingly popular. Take smartwatches, arguably the most important type of smart wearables, as an example. According to a market research report published recently [9], the global market value of smartwatches was estimated to be $10.2 billion in 2017 and will experience an annual growth rate of 22.3% from 2018 to 2023.

In the literature, several efforts have been made towards understanding and improving the OS execution performance [48, 49], power management [51], graphics and display [54], storage [37], and user interface [20, 78] of wearable OSes. In this paper, we investigate an important yet under-explored component: *the wearable networking stack*. We conduct to our knowledge a first in-depth investigation of the networking performance of Wear OS, one of the most popular OSes for wearables. Wear OS is a version of Google's Android OS tailored to small-screen wearable devices. Used

Authors' addresses: Xiao Zhu, University of Michigan, Ann Arbor, MI, USA, shawnzhu@umich.edu; Yihua Ethan Guo, Uber Technologies, Inc. San Francisco, CA, USA, guoyihua@uber.com; Ashkan Nikravesh, University of Michigan, Ann Arbor, MI, USA, ashnik@umich.edu; Feng Qian, University of Minnesota – Twin Cities, Minneapolis, MN, USA, fengqian@umn.edu; Z. Morley Mao, University of Michigan, Ann Arbor, MI, USA, zmao@umich.edu.

2476-1249/2019/3-ART3 $15.00
https://doi.org/10.1145/3311074

by a wide range of smartwatches and potentially other wearables, Wear OS is expected to account for 41.8% of the market share of smartwatch OSes in 2020 [7].

Wearable networking is *important*. Take smartwatches as an example. One may argue they only incur light traffic such as push notifications. This might be true for the *current* smartwatch ecosystem where traffic flows are largely small, short, and bursty [51]. However, we envision that future wearable apps will be more network-intensive by incurring much heavier network activities as fueled by new hardware, OS support, and applications. For example, recently debuted speaker/LTE-capable watches such as LG Watch Urbane 2nd Edition allow users to directly make hands-free VoIP calls; the latest Wear OS 2.x allows standalone apps on wearables; also, many emerging wearable applications incur heavy network traffic such as continuous computer vision on smart glasses [23, 31], remote camera preview [6], real-time screen projection [2], and network-level collaboration between phone and watch [52].

Wearable networking is also *different* from smartphone networking that has been well studied in the past decade. First, wearables oftentimes do not directly access the Internet; instead, it uses its paired smartphone as a "gateway", which, if not carefully designed, may incur additional performance degradation. Such a gateway mode accounts for 84% of the daytime usage period as measured by a recent user study [51]. Second, the communication between a wearable and the phone is usually through Bluetooth (BT) or Bluetooth Low Energy (BLE), whose characteristics are vastly different from WiFi and cellular that dominate the smartphone interface usage; also the cross-layer interaction between BT and upper-layer protocols such as TCP remains underexplored. Last but not least, due to BT's short range, network handovers frequently occur on a wearable: when it moves away from the phone, the BT connectivity will be torn down and the wearable has to use standalone WiFi or LTE to communicate with the external world.

Understanding the networking performance of commercial wearables is challenging, as it involves multiple devices, networks, and protocols, which incur complex interactions. The proprietary nature of Wear OS makes it even harder to gain deep visibility into the wearable networking stack. Note that unlike Android for handheld devices, Wear OS is not open-source.

To address these challenges, we first build a wearable networking testbed consisting of commodity Wear OS based smartwatches, off-the-shelf smartphones, commercial wearable apps, as well as a series of tools we developed for instrumenting the system and collecting various types of data. We then leverage the testbed to conduct controlled experiments in a cross-device, cross-protocol, and cross-layer manner. Through judiciously designed experiments, we demystify the Wear OS networking stack and quantify how it affects the wearable networking performance. Our key findings consist of *several serious performance issues* regarding *all* three aforementioned aspects that distinguish wearable networking from smartphone networking.

- We perform a comprehensive analysis of the BT radio state machine on both the wearable and its paired phone. We find tricky yet critical differences of the state machine behaviors between the two sides. They lead to our key discovery that due to the wearable's unique BT radio management policy and its interplay with its counterpart on the phone, a download session on the wearable frequently (*e.g.,* every few seconds) experiences "blackout" periods lasting for about 1 second (§3).
- When acting as a gateway proxy for a wearable, the phone dramatically inflates the end-to-end (server to wearable) latency to 30+ seconds due to its incurred "bufferbloat". We then break down the end-to-end latency into various components, and identify the root cause to be the phone-side TCP receive buffer, whose configuration does not take into account the path asymmetry between the wearable-phone path and the phone-server path (§4).
- Wearables are equipped with multiple network interfaces such as BT and WiFi. When multiple networks are available, the Wear OS's default interface selection policy strictly prefers one interface (*e.g.,* BT) over others (*e.g.,* WiFi). However, we find that such a strategy oftentimes leads to

Table 1. Study summary: examined aspects, methodologies, key findings, root causes, and recommendations.

Aspect	Methodologies	Key findings	Root causes of inefficiency	Recommendations
BT radio resource management (§3)	Cross-device state machine inference.	Different state machine models on phone and wearable; BT download experiences frequent "blackout" periods.	Undesired BT sniff (sleep) mode is triggered during a continuous download.	BT state machine should be aware of bidirectional traffic and judiciously determine when to enter the sniff mode.
Proxying at paired smartphone (§4)	Breakdown analysis of end-to-end latency.	Phone-side bufferbloat inflates the end-to-end latency to tens of seconds; real-time apps' performance is severely affected.	Multiple buffers at different layers on the smartphone proxy, in particular the large TCP receive buffer, cause the high buffering delay.	The proxy should be aware of the path asymmetry between server-phone and phone-wearable paths.
Network interface selection (§5)	Quantify energy-performance tradeoffs using real wearable apps; develop a multipath framework for wearables.	Wear OS's default interface selection policy is often suboptimal; BT+WiFi multipath brings limited performance gain.	Static interface preference is not aware of apps' QoE requirements; BT and 2.4 GHz WiFi cause interference.	Need QoE-aware interface selection; need 5GHz WiFi support on wearables; need BT-aware multipath scheduler.
Network handover (§6)	Conduct an IRB-approved user study; examine each phase of a handover process.	Frequent BT-WiFi handovers in the wild; short BT range on commodity wearables; BT-WiFi handovers may take 60+ seconds.	Handovers are performed reactively instead of proactively; both OS and app logic contribute to high handover delay; lack protocol/OS support for handover.	Proactively predict a handover; need multipath support to facilitate seamless handovers.

suboptimal tradeoffs between performance and energy consumption. In addition, we explore the feasibility of performing multipath transport (simultaneously using WiFi and BT) on wearables, and identify potential obstacles such as the interference between BT and 2.4GHz WiFi (§5).

• BT's short communication range makes handovers occur frequently on wearables. Due to insufficient protocol support and poor cross-layer coordination, a BT-WiFi handover may last for more than 60 seconds, leading to significant disruption of the wearable application performance. By looking into each phase of a handover, we find that both the OS and user application are responsible for such unacceptably long handover delays (§6).

The above performance inefficiencies are caused by the poorly designed networking stack of Wear OS. Our identified issues appear on all 8 wearables of heterogeneous vendors and Wear OS versions (including the latest version as of December 2018) as well as a variety of paired phones as tested by us using synthetic and real apps. To mitigate the identified performance impairment, we design, implement, and evaluate several readily deployable mitigation solutions including the following. (1) We develop a lightweight module that completely eliminates the undesired behavior of the wearable's BT radio state machine (§3.3), (2) We develop a simple yet effective flow control scheme that mitigates the phone-side bufferbloat problem, achieving up to 78x latency reduction with less than 3% of the throughput decrease (§4.3), (3) We design and implement to our knowledge a first multipath transport framework for wearable devices that enables adaptive interface selection, multi-network bandwidth aggregation (§5.2), and smooth handovers between IP and non-IP networks (§6.4). For example, our improved handover scheme reduces the BT-to-WiFi handover delay from more than 28 seconds to less than 0.6 seconds with negligible energy overhead incurred. Table 1 summarizes the key findings made in this paper. Note that while some identified issues such as those incurred by the BT state machines can be relatively easily fixed, tackling

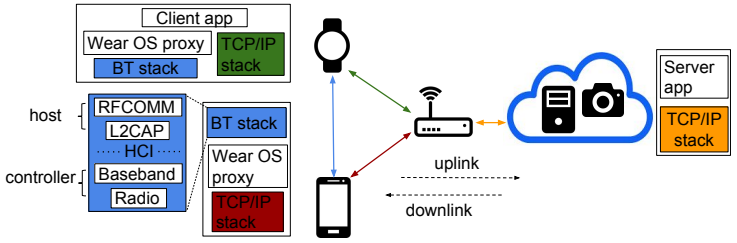


Fig. 1. The measurement testbed (middle) and the protocol stacks of the wearable, phone, and server (left and right).

other aspects such as bufferbloat, handover, interface selection, and multipath for wearable devices requires considerable efforts due to various challenges in the wearable ecosystem as mentioned earlier. We believe our work initiates this research thrust, and more future research is needed along this direction.

Overall, this paper makes contributions in three aspects. (1) We develop novel methodologies for measuring and analyzing the wearable networking performance. (2) We discover severe and previously unknown performance issues of Wear OS's networking stack. (3) We identify their root causes and application performance impact, and propose corresponding mitigation strategies.

2 BACKGROUND AND METHODOLOGY

The wearable networking is unique in several aspects, making analyzing its performance and resource consumption challenging.

• Instead of accessing the Internet directly, a wearable typically leverages a paired mobile device such as a smartphone as a gateway.

• Compared to performing pure TCP/IP networking on a regular host, wearable networking involves both BT and TCP/IP. In particular, since BT by default does not speak TCP/IP, the wearable OS typically introduces a pair of proxies on the smartphone and the wearable to bridge TCP/IP and BT. For the phone-side proxy, it maintains TCP connections to remote servers on behalf of the wearable. It strips off TCP/IP (BT) headers for downlink (uplink) traffic, and encapsulate the application data into BT (TCP/IP) packets. A reverse operation is performed at the wearable-side proxy, which also maintains local TCP connections with client apps.

• The BT protocol stack itself is complex. It consists of higher-layer protocols realized in the *host* (software) and lower-layer functions implemented in the *controller* that resides on the BT chip. The host and controller are bridged by the Host-Controller Interface (HCI). The BT performance can thus be affected by multiple factors at different layers as well as its interplay with TCP/IP and the aforementioned proxying mechanism.

• Wearable OS developers usually keep their implementation proprietary. Unlike Android for handheld, Wear OS is not open-source.

To address the above challenges, our high-level approach is to develop a holistic testbed and a suite of measurement tools that comprehensively examine not only each of the aforementioned components, but also the cross-device, cross-protocol, and cross-layer interplay on real wearables over real wearable apps' workload. We next describe our testbed and measurement toolkit design.

2.1 Wearable Networking Testbed

We set up a testbed shown in Figure 1 to cover common usage scenarios for a wearable to communicate with the external world. They include communicating locally with the phone over BT, accessing the Internet directly with WiFi/LTE, as well as surfing the Internet via the smartphone

Table 2. Mobile devices used in our experiments.

Smartwatch Model	Wear OS Version	Paired Smartphone	Smartphone Android OS
LG Urbane	1.5	Nexus 5	6.0.1
LG Urbane	2.15	Nexus 5X	7.1.1
LG Urbane 2nd Edition	2.0	Samsung Galaxy S5	5.1.1
LG Urbane 2nd Edition	2.20	Pixel 2	9.0.0
Huawei Watch	2.0	Nexus 6P	7.0.0
Huawei Watch 2	2.9	Nexus 5X	7.1.1
Asus ZenWatch 3	2.0	Nexus 5	6.0.1
LG G Watch R	2.0	Nexus 5	6.0.1

as the gateway (called the *CPROXY* mode in Wear OS). Our testbed consists of 8 state-of-the-art smartwatches listed in Table 2. All of them support BT and WiFi while some higher-end watches such as LG Urbane 2nd Edition and Huawei Watch 2 support LTE as well. The OSes we study include the latest release (Wear OS 2.20 released in December 2018) as well as the older Android Wear OSes 2.x and 1.x. Our measurement findings apply to all OSes unless otherwise mentioned. The Internet server we use is equipped with a quad-core 3.6GHz CPU and 16GB memory, running Ubuntu 16.04. We run on the testbed the workload generated by our measurement tools (described shortly) and real apps that perform bulk data transfer, constant bitrate transfer, and real-time streaming. We also employ a Samsung SNH-V6414BN SmartCam to stream real-time video to smartphones and smartwatches.

2.2 The Wearable Network Measurement Tools

Given a lack of tools for measuring and analyzing wearable network performance especially over BT, we also develop a suite of tools to fill this gap. They consist of software programs for both active and passive measurements. We will use them to conduct carefully crafted black-box testing without requiring the OS source code. This is to our knowledge the most comprehensive toolkit for wearable networking performance analysis and diagnosis.

For active measurements, we develop a custom server application running on the server and a custom client app running on the wearable. Supporting all aforementioned communication paradigms, the client and server apps can exchange data using two traffic patterns: bulk data transfer and constant bitrate over the uplink (from the wearable), downlink (to the wearable), or both. Our application also allows automatic reconnection upon network failure for testing the handover support, an important feature needed for wearables due to their short BT range (§6).

For passive measurements, we collect both WiFi and BT traces on multiple entities (phone/wearable/server). The BT trace is captured at both the host-controller interface (HCI, using `btsnoop` log) and the OS (using `tcpdump`), and contains both the data packets and the BT control messages. In addition to the network traces, we collect the network state and signal strength information to understand their impact on network performance. We also develop a tool that can instrument different components of the packet transmission/reception pipeline in the OS kernel to identify the performance bottleneck for the end-to-end data delivery (§4.2).

Compared to prior measurement studies, our measurement and instrumentation techniques are comprehensive in that they cover multiple entities (wearable, phone, server), protocols (BT, WiFi), and instrumented layers (HCI, OS, TCP, App). Note this may not be the case for many prior works. For instance, some previous studies on smartwatches [51, 79] collect BT traces only at HCI, incurring various limitations such as inaccurate goodput measurement (due to lower-layer padding) and not being able to separate individual application streams from the multiplexed traffic captured

Table 3. Data collected by our measurement toolkit.

Category	Data Item	Method	Source
Watch-side	BT HCI trace	Callback	`btsnoop` log
	BT and WiFi packet trace		`tcpdump`
	BT RSSI	Poll (0.2s)	Wear OS API
	BT and WiFi network state		
Phone-side	BT HCI trace	Callback	`btsnoop` log
	TCP/IP packet trace		`tcpdump`
	kernel packet transmission events		Kernel log
	BT packet trace		Android log
Server-side	packet trace	Callback	`tcpdump`

at HCI. Some other methods [29, 41, 73] extract the RTT from only one side, and are therefore incapable of inferring the end-to-end RTT when a wearable–server connection is split by a phone when the *CPROXY* mode is used. Table 3 lists all types of data collected by our toolkit. The runtime CPU overhead of collecting those types of data is less than 3% on our wearables.

The collected data will be analyzed offline. Given a lack of tools to decode BT messages, we follow the BT specification [10] to build a tool that can parse the BT traffic to extract both the user payload and control messages. In addition, this offline tool can perform various types of correlation analysis on different data sources, including cross-technology (e.g., WiFi vs. BT), cross-device (e.g., wearable vs. phone), and cross-layer (e.g., app performance vs. BT radio state) correlation analysis. Our toolkit is written in about 3,000 LoC using C++, Java, and Python. We have open-sourced the entire toolkit on GitHub [12].

Leveraging the above measurement infrastructure, we next answer the following important research questions.

- What is the BT performance on commodity wearables? How is the upper-layer wearable app performance affected by the lower-layer BT resource management state machine?
- How does the smartphone gateway impact the performance?
- How does the network selection policy affect the tradeoff between performance and energy consumption?
- What is the performance when a network handover occurs?

3 IMPACT OF BT RADIO STATE MACHINE

To reduce the energy consumption and efficiently utilize the limited radio resources, wireless radios usually define different radio states to operate on [38]. BT makes no exception. In this section, we investigate how the BT radio state machine behaves on commodity wearables, and quantify its impact on wearable networking performance. This is to our knowledge the most comprehensive and in-depth study of the BT radio state machine on wearables. We discover that the poorly realized BT state machine on all Wear OS smartwatches we have incurs significant performance degradation in common usage scenarios (*e.g.,* up to 140% inflation of the data download time over BT).

3.1 BT Radio State Primer

The BT core specification [10] defines four radio states: the Active mode, Sniff mode, Hold mode, and Park mode. In the Active mode, the device is always listening for packet transmission and reception, while the latter three are low-power modes where the BT device sleeps during most of the time, and periodically wakes up to listen to the channel to check if there is any incoming data. A state transition can be triggered by a BT device itself, or by requests from a peer device. In the

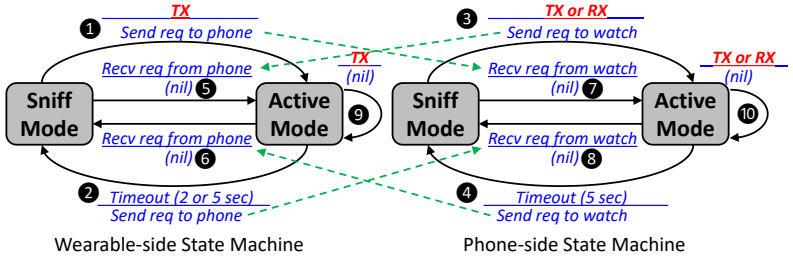


Fig. 2. BT state machines on wearable and phone. A state transition is described by its *condition* (above the bar) and its incurred *action* (below the bar). "nil" means no action.

latter case, to request a state transition, a BT host can issue HCI commands to its BT controller, which will further notify the peer node (see the left side of Figure 1 for the BT protocol stack). The peer can choose to accept or reject this state change request, and/or to suggest a different set of parameters. Each side's BT host will be notified with the state change request or response through HCI. The radio states and their transitions can incur tradeoffs between the performance and energy consumption. Note that other types of wireless radios such as WiFi and cellular also have radio state machines, and they have been well studied in the literature [19, 56]. In contrast, BT state machines receive much less attention in particular on wearables.

3.2 Inferring the BT Radio State Machines

Despite defining different radio states, the standard does not specify the actual state transitions, which are up to vendors' implementation choices. Also, a unique aspect of the BT state machine that was not considered by the prior study [79] is that a device's radio state is jointly determined by itself and its paired device as described in §3.1. We thus perform a study to comprehensively infer the BT radio state machine for off-the-shelf wearables and phones, as well as to study their interplay. Our high-level approach is to conduct controlled experiments using strategically crafted traffic (by controlling the direction, size, and timing of the traffic) to exercise different states and the transitions among them. Also, instead of indirectly tracking the state transitions (*e.g.,* using power measurement [79]), we use our tool (§2) to capture and analyze the BT control messages at the HCI layer to explicitly and precisely monitor the state transitions without requiring a power monitor. We next detail our approach for inferring the BT state machines on both the wearable and the phone as shown in Figure 2.

• **State Simplification.** To begin with, through extensively testing against various workloads, we identify that among the four radio states defined by the specification, none of our 8 smartwatches or 8 smartphones listed in Table 2 uses the Hold mode or the Park mode. We thus only need to consider the transitions between two modes (states): the Active and the Sniff mode.

• **Phone-side State Transitions.** We now describe how a device determines state transitions based on its local state and its observation of the network traffic. We first examine the phone side. Through controlled experiments, we confirm that a Sniff→Active transition is triggered by any data to be transmitted or received (❸ in Figure 2). Recall that at the Sniff mode, the device typically sleeps, and only periodically wakes up to check incoming data. Therefore, efficient data transmission and reception have to be performed on the Active mode. We also confirm that an Active→Sniff transition (❹) is triggered by an inactivity timer, whose value is 5 seconds based on our measurements. The timer is reset whenever any data is sent or received at the Active mode (❿).

• **Wearable-side State Transitions.** We apply the same method to infer the wearable-side state transitions, which (❶, ❷, and ❾) are found to be the same as those at the phone side except two differences. First, very surprisingly, by testing the state machine's behaviors using uplink, downlink,

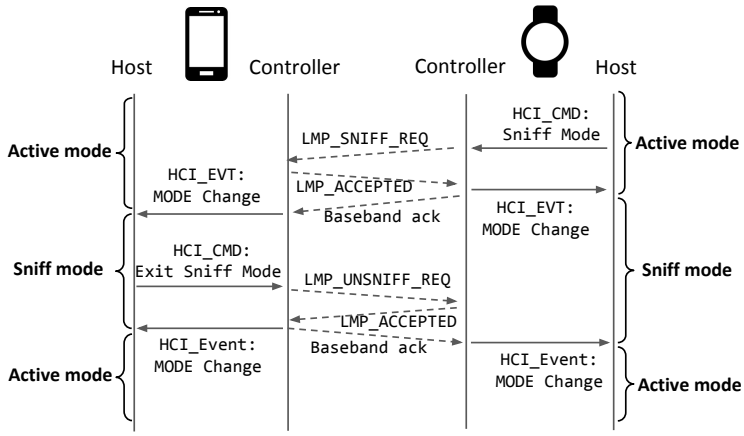


Fig. 3. BT control message exchanges for triggering and exiting Sniff Mode.

and bidirectional traffic, we discover that for all 8 watches, a Sniff→Active transition is *only triggered by transmissions but not by receptions* (❶); also the Active mode timer is *only reset by transmissions but not by receptions* (❾). As a result, if a wearable keeps receiving data but not transmitting any data, it will enter the Sniff mode when the timer expires. We call this *Undesired Sniff mode* problem. We will revisit it shortly and demonstrate its severe performance impact in §3.3. Second, we observe two timers, 2s and 5s, that might be used by the wearable's Active→Sniff transition (❷). As to be shown soon, the 2s timer worsens the undesired Sniff mode issue compared to the 5s timer.

• **Interplay between Phone and Wearable.** We next describe how the phone-side state transitions affect the wearable-side ones, and vice-versa. Recall from §3.1 that the BT state machine has a unique mechanism: when its own state is about to change, a device A will send a request message to its paired device B to request B's state to also be changed, in order to *explicitly* synchronize both sides. The dotted arrows in Figure 2 illustrate such message deliveries. Per the BT specification, B can either accept or decline this request; but our measurements indicate that in practice, the paired device (either the wearable or the phone) will always accept the request, as shown in ❺, ❻, ❼, and ❽ in Figure 2.

Figure 3 illustrates the above process and how it interplays with the aforementioned undesired Sniff mode problem on wearables. Let us assume that an on-going download is in progress so both devices are in the Active mode. We describe four sequentially occurred events. (1) Due to the undesired Sniff mode problem, the received data does not reset the Active mode timer on the wearable side, so the wearable will eventually enter the Sniff mode (❷), and send a request to the phone (`LMP_SNIFF_REQ` in Figure 3) to let the phone enter the Sniff mode as well. (2) As mentioned above, the phone unconditionally accepts this request and enters the Sniff mode (❽). (3) However, since the phone is still transmitting data, a Sniff→Active transition will be immediately triggered on the phone side (❸, recall that the phone does not have the undesired Sniff mode problem), which also sends a request to the wearable to change its state. (4) Through this way, the wearable exits the Sniff mode (❺) and resumes the data reception. However, this process will repeat after a while, based on the wearable's Active mode timer.

Figure 4 illustrates such repetitions during a long-lived data download session (monitored on the wearable side). The green triangles are phone-originated requests received by the wearable (*i.e.,* the trigger of ❺). The red squares represent the Sniff mode requests sent by the wearable (*i.e.,* the action of ❷). As shown, when there is no uplink traffic, between a triangle and the next square, there is a constant interval of 5 seconds corresponding to the wearable's Active mode timer. Such

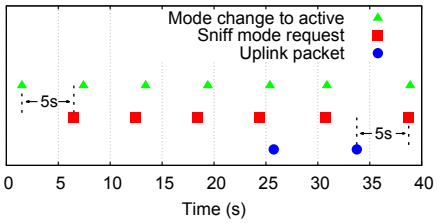


Fig. 4. Repetitive transitions between Sniff and Active modes observed on LG Urbane (normal RSSI).

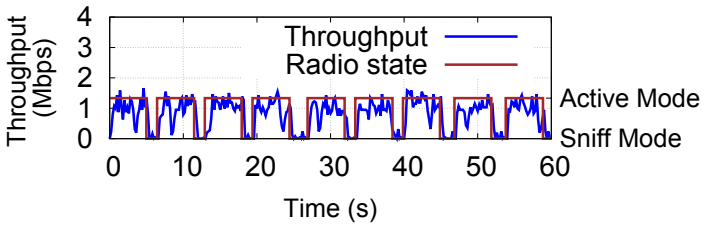


Fig. 5. Correlation between the BT throughput and radio state for download, demonstrating the undesired Sniff mode problem (LG Urbane Watch, normal RSSI).

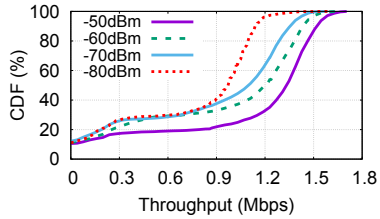
an interval can be prolonged by uplink packets that reset the wearable's Active mode timer, as indicated by the blue dots in Figure 4.

3.3 Impact of the Undesired Sniff Mode

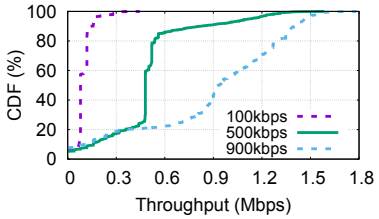
We believe that it is worthwhile reporting the undesired Sniff mode problem due to two reasons. First, this problem is prevalent as it has been observed on *all* of our 8 wearables of heterogeneous vendors and Wear OS versions. Second, the Wear OS is a tailored version of Android yet their state machines behave differently. This may reflect several *incorrect* assumptions possibly made by BT driver developers for wearables:
× The Sniff mode always helps reduce energy consumption without incurring much performance degradation. So it should be aggressively used.
× The duration of the undesired Sniff mode is short as it can be quickly recovered by the state change requests from the phone.
× Wearables only receive little data, or the wearable traffic is always a mixture of downlink and uplink, so the undesired Sniff mode is unlikely to occur.

We next experimentally demonstrate the severe performance impact brought by the undesired Sniff mode by examining two performance metrics: throughput and one-way delay (OWD) during a long-lived data download session. The throughput is calculated every 200ms on the receiver (wearable) side. The OWD from the server to the wearable is an important performance metric for real-time applications. It is continuously measured as the difference between the transmission and the reception time of each byte. Before each experiment, we connect the wearable through a USB cable to the server, and use a custom program we developed to synchronize their clocks. The results below are obtained on an LG Urbane smartwatch running the Wear OS 2.15 paired with a Nexus 5X running Android 7.1.1. Other wearables and phones show qualitatively similar results.

In Figure 5, we show the performance of the bulk data download, which represents important use cases such as downloading apps, software update, or media files to a wearable. As shown, the wearable's radio state oscillates between the Active and the Sniff mode during a continuous download. As a result, for every ~5 seconds (the wearable's Active mode timer), the BT throughput

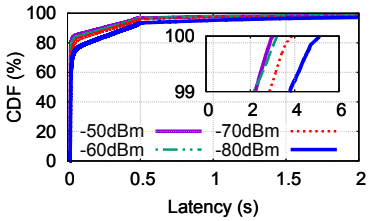


(a) Bulk download under different phone-watch positions.

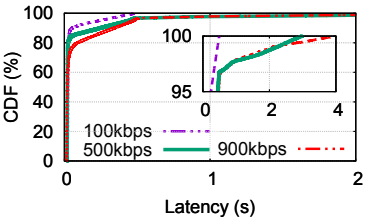


(b) Different data rates of CBR traffic, -50 dBm BT RSSI.

Fig. 6. BT downlink throughput of bulk data transfer and CBR on LG Urbane Watch.



(a) 500kbps CBR under different phone-watch positions.



(b) Different data rates of CBR traffic, -50 dBm BT RSSI.

Fig. 7. End-to-end OWD for CBR traffic on LG Urbane.

drops to almost 0 for about 0.5 to 1.2 seconds. A similar observation is made for CBR streaming, which represents real-time traffic such as VoIP and live radio streaming (figure not shown).

To further quantify the impact of the undesired Sniff mode in diverse environments, we repeat the above bulk data download experiment by varying the location of the smartphone: (1) on a desk where the user sits in front (-50 dBm); (2) in the user's pocket (-60 dBm); (3) in the user's bag (-70 dBm); (4) meters away from the watch (-80 dBm). Note that the smartwatch is always worn on the user's wrist. As shown in Figure 6a, in all the settings, we observe throughput drop and at least 10% of the no-reception time. We also use three different data rates for CBR traffic (-50 dBm BT RSSI) and observe severe throughput degradation for all three rates, as shown in Figure 6b.

We next measure the impact of the undesired Sniff problem on the OWD for CBR traffic. Similar to Figure 6a, Figure 7a measures the OWD for CBR traffic at 500 kbps under four settings with different BT RSSI readings. The long tails indicate that about 30% of the OWD samples are affected. The OWD can inflate to up to 5 seconds. A similar observation is made in Figure 7b, which measures the OWD distributions for downlink CBR traffic at three data rates with -50 dBm BT RSSI.

Mitigation Solutions. The ultimate fix of the undesired Sniff mode problem requires modifying the wearable-side state machine residing in the lower protocol stack. Nevertheless, the results in Figure 4 suggest a simple temporary fix: generating light uplink traffic during a download session. We implement this solution by developing a lightweight background app running on a wearable. It tracks the wearable's BT radio state based on the observed traffic, and sends a small uplink packet when the wearable's Active→Sniff timer is about to expire. We find that this simple solution completely eliminates the undesired Sniff mode and therefore its incurred performance degradation.

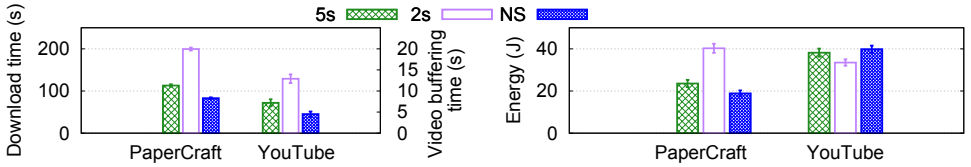


Fig. 8. QoE (left) and energy consumption (right) for game download and YouTube streaming on LG Urbane.

Real App Performance. We also conduct tests on real apps. We confirm that the undesired Sniff mode problem does occur on commercial wearable apps and indeed affects the user experience. We run two apps under good BT network condition (-50 dBm RSSI): downloading the PaperCraft game app of 16 MB from the Google Play store and watching a YouTube video[1]. The left plot in Figure 8 shows the app QoE for the two workloads: the overall download time for the app download (the left Y axis) and the initial video buffering time (the right Y axis). We show three bars for each app. "5s" and "2s" correspond to having a wearable-side Active→Sniff timer of 5s and 2s, respectively. "NS" refers to the scenario where the undesired Sniff mode problem is fixed using the above solution of dynamically injecting light uplink packets. The right plot in Figure 8 shows the energy consumption in each case. We calculate the energy consumption using a full-fledged smartwatch energy model developed recently [51]. Each experiment is repeated 10 times. We make two observations. First, the undesired Sniff mode brings significant QoE degradation to both apps. Compared to the 5s Active→Sniff timer, fixing the undesired Sniff mode issue reduces the app download time and the initial video buffering delay by 27% and 37%, respectively. Compared to the 2s timer, the gains are even higher – 58% and 65% respectively. Second, entering the Sniff mode more frequently is supposed to bring energy savings. However, for app download, it actually increases the overall energy consumption by 25% and 113%, when the timer is 5s and 2s, respectively, because the overall download time is lengthened. For YouTube video streamed at a lower bitrate (compared to the file download), eliminating the undesired Sniff mode only slightly increases the energy consumption (by 4% when the timer is 5s).

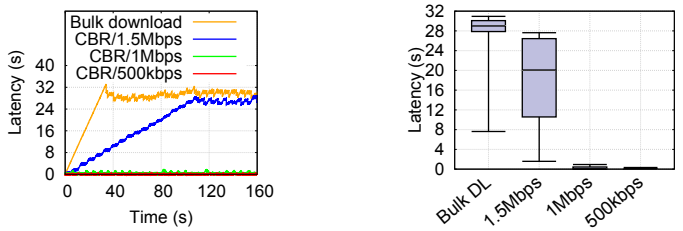
4 IMPACT OF SMARTPHONE PROXYING

As mentioned earlier, a paired smartphone gateway plays a critical role in wearable networking. In this section, we study the performance impact of the *CPROXY*. Recall that typically residing on a paired phone, the *CPROXY* splits an end-to-end client-server connection into a server-phone TCP connection and a phone-wearable BT RFCOMM connection, while being transparent to both the wearable-side and server-side apps. Because of the two heterogeneous links, the *CPROXY* needs multiple buffers at various layers, such as the receive buffer in the TCP/IP stack, the app-layer buffer, and the transmission buffers in the BT RFCOMM stack. These buffers, along with other existing in-network and on-device buffers, can potentially cause "bufferbloat" that inflates the end-to-end delay. This is particularly undesired for real-time traffic with low latency requirements.

4.1 Substantial Bufferbloat in *CPROXY*

We begin with characterizing the overall end-to-end latency under the *CPROXY* mode. Specifically, we measure the one-way delay (OWD) from the server to the wearable, an important performance metric for real-time applications. To emulate the real-time traffic, we use the CBR traffic with three rates as the workload: 1.5Mbps, 1Mbps, and 500kbps. For comparison, we also measure the OWD of bulk data download without a rate limit.

[1]Video streaming on smartwatches has its use cases such as watching a tutorial when performing cooking or house maintenance. The playback can be controlled by voice.



(a) E2E delay over time. (b) E2E delay distribution.

Fig. 9. E2E delay of bulk transfer and CBR traffic (LG Urbane paired with Nexus 5X, normal RSSI).

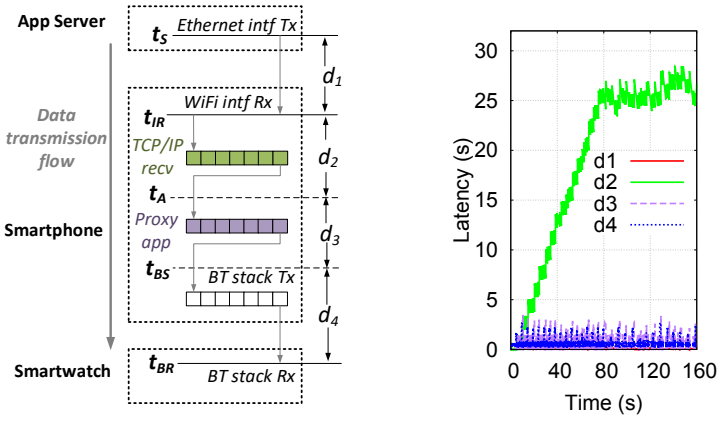
For each sending rate, the server sends data to the wearable for at least 160s. The OWD is measured using the method introduced in §3.3.

Figure 9 shows the OWD of CBR traffic and the bulk download from the server to an LG Urbane Watch paired with a Nexus 5X over one representative measurement. For CBR traffic whose data rate is much lower than the BT bandwidth, we still observe fluctuating OWD over time, with the standard deviation being 324ms (99ms) for 1Mbps (500kbps). When the CBR rate becomes higher, *e.g.,* at 1.5Mbps, the OWD inflates to an unacceptably high level, with the median delay being 20.1s (up to 28.6s). For bulk download, its median OWD further increases to 29.0s. We also observe high delays on other combinations of watches and phones we have. Recall from §1 that many wearable apps incur high-bitrate real-time traffic, such as real-time camera streaming, HD VoIP, and real-time screen projection [2]. The high OWD will incur unacceptable QoE for such apps.

4.2 Identifying the Root Cause

We now seek to understand the root cause of the high OWD under the *CPROXY* mode. The multiple buffers scattered in the end-to-end data transmission pipeline present a challenge towards our analysis. We thus dissect the end-to-end (E2E) delay by instrumenting at multiple entities and layers. Specifically, we use our toolkit (Table 3, §2) to collect BT and TCP/IP traces at several locations, and then perform offline analysis to obtain for each byte various timestamps as illustrated in Figure 10. (1) t_S: from the `tcpdump` trace captured on the server when the data is being transmitted out; (2) t_{IR} from the `tcpdump` trace captured on the smartphone when the data is received in the smartphone OS kernel; (3) t_A: from the kernel log captured on the smartphone when the data is copied to the proxy app's userspace (by instrumenting `tcp_input.c`); (4) t_{BS}: from the Android log captured on the smartphone when the proxy app sends the data to the BT stack (by instrumenting `BluetoothSocket.java`); (5) t_{BR}: from the `tcpdump` trace on the wearable when the data is delivered to wearable OS. The end-to-end latency can thus be broken down into four parts: the transmission delay from server to phone ($d_1 = t_{IR} - t_S$), the buffering time in the TCP/IP stack on the phone ($d_2 = t_A - t_{IR}$), the buffering time in the proxy app buffer on the phone ($d_3 = t_{BS} - t_A$), and the delay of BT transfer from the phone to the wearable ($d_4 = t_{BR} - t_{BS}$). Note that d_2 is dominated by the delay incurred by the TCP receive buffer on the smartphone. The IP queueing delay at the `qdisc` is confirmed to be very small. Also, we separate d_2 and d_3, both residing on the smartphone, due to the difference between their associated buffers: the TCP buffer incurring d_2 is maintained at a per-connection basis, whereas the proxy app buffer incurring d_3 is shared by all wearable app streams, and is therefore more likely to cause potential cross-traffic interference.

Measurement Results. Figure 10b shows the OWD breakdown for CBR traffic at 1.5Mbps for an LG Urbane watch paired with a Nexus 5X, over a representative experiment. We observe that the buffering delay in the TCP/IP stack (d_2) accounts for almost the entire OWD. Recall that d_2 is



(a) E2E delay breakdown methodology.

(b) Breakdown of E2E delay for CBR traffic at 1.5Mbps.

Fig. 10. E2E delay breakdown of CBR traffic in *CPROXY* mode (LG Urbane with Nexus 5X, normal BT RSSI).

Table 4. Impact of TCP receive buffer size on the severity of *CPROXY* bufferbloat on different phones.

	Nexus 5X	SGS5	Nexus 5
`tcp_rmem_max`	8,291,456	4,525,824	2,097,152
`rmem_max`	8,388,608	2,097,152	2,097,152
d_2: TCP/IP recv (s)	26.1 ~ 28.6	4.0 ~ 5.5	4.1 ~ 5.7
Total E2E OWD (s)	27.9 ~ 30.1	5.7 ~ 6.7	5.9 ~ 7.0

dominated by the delay incurred by the TCP receive buffer (recvBuf). We thus explicitly confirm how the recvBuf size affects the OWD on three smartphones in Table 4. The *effective* recvBuf size is determined by the minimum value of two configurable OS parameters `rmem_max` and `tcp_rmem_max` (both are in bytes). As shown, a phone with a smaller recvBuf indeed experiences a smaller d_2 as well as a lower overall E2E OWD. However, setting the recvBuf to be too small will throttle the TCP congestion window and hence the throughput – a tradeoff that is difficult to balance.

While the bufferbloat problem has been well studied in different contexts such as broadband wired network [74], cellular download [41], and cellular upload [29], we highlight two differences that make bufferbloat in the *CPROXY* mode a unique problem. First, due to the highly asymmetric bandwidth of the BT/BLE link and the WiFi/cellular link, the *CPROXY*-side bufferbloat will always occur when the WiFi/cellular link throughput becomes higher than ~1.1Mbps. The above breakdown analysis indicates that the TCP recvBuf configuration does not take into account such bandwidth asymmetry. Second, the lower-layer BT state machine also affects the severity of the bufferbloat. In particular, the undesired Sniff Mode identified in §3.2 slows down the BT data transmission and thus causes the proxy-side buffer to further build up. This is confirmed in Figure 10b where d_4 exhibits periodical spikes, whose occurrences well match those of entering the sniff mode.

4.3 Mitigating the *CPROXY* Bufferbloat

We now consider how to mitigate the *CPROXY* bufferbloat. In the literature, numerous bufferbloat mitigation solutions have been proposed, but we found it is difficult to directly apply them in our context due to various practical or fundamental issues. For example, blindly reducing the TCP recvBuf may throttle the congestion window and thus the throughput [29]; delay-based TCP

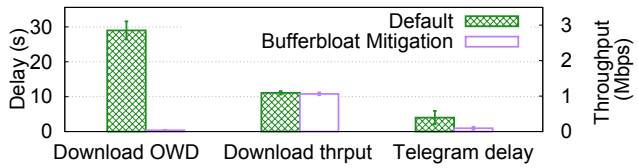


Fig. 11. *CPROXY* bufferbloat mitigation for bulk download and messaging with competing traffic (LG Urbane, Nexus 5X).

congestion control [16, 50] is not aware of the BT protocol stack between the phone and the wearable; various Active Queue Management (AQM) techniques [55, 63] only regulate the qdisc buffering, and may need substantial modifications for tackling the *CPROXY*-side bufferbloat.

Developing a full-fledged bufferbloat mitigation solution for wearable networking with heterogeneous links is beyond the scope of this paper. Here, we propose a simple, practical, yet effective solution to demonstrate the need for coordinating the heterogeneous links as well as the substantial performance improvement. Note that other (better) solutions may exist.

In our scheme, the phone maintains a virtual queue (shared by all apps) whose size increases as bytes arrive from the remote server and decreases upon the reception of BT ACKs. Based on the virtual queue size, our scheme dynamically throttles the connection between the phone and the server (if needed) to bound the actual buffering delay. Specifically, we maintain two thresholds, an upper bound Q_{UB} and a lower bound Q_{LB}. The throttling is enabled when the buffer level exceeds Q_{UB}, and is disabled when the buffer level drops below Q_{LB}. Q_{UB} is set to $BW \times (1-\varepsilon)T$ where BW is the current estimation of the BT link bandwidth, T is the upper bound of the tolerable queueing delay (configurable based on the app's QoE requirement), and ε controls the aggressiveness of our scheme. Q_{LB} is set to $BW \times (1-2\varepsilon)T$ so that both thresholds are proportional to the BT link bandwidth. We empirically use T=1s, ε=0.3, and set the throttling rate to $\frac{BW}{2}$. Note that BW may vary over time.

Evaluation. We implement the above scheme using our toolkit (§2) for performance monitoring and Linux `tc` for bandwidth throttling. We then conduct controlled experiments to evaluate its effectiveness. We consider two workloads: TCP bulk download and receiving short messages delivered by the Telegram messaging app [3] when there is an on-going concurrent transfer. The latter scenario may happen when, for example, a user receives a message when a media player is performing audio or video streaming in the background. We repeat both experiments 10 times under a normal network condition (-60 dBm BT RSSI) on an LG Urbane smartwatch paired with a Nexus 5X phone. Figure 11 measures the OWD and throughput for the bulk download, as well as the per-message delivery time for Telegram messaging. As shown, for bulk download, our scheme substantially reduces the packet OWD by 78 times with less than 3% of throughput reduction. Our scheme also reduces the Telegram message delivery delay by 76%.

5 PERFORMANCE & ENERGY IMPACT OF NETWORK SELECTION

Today's wearables are usually equipped with multiple network interfaces. For example, most smartwatches have WiFi and BT/BLE, and advanced editions even have the cellular interface [44]. Typically, the Wear OS employs a *static* interface selection policy: all 7 smartwatches except Huawei Watch 2 use BT (through the *CPROXY*) when both BT and WiFi networks are available. At first glance, this simple policy is energy-wise beneficial as BT is known to be more power-efficient than WiFi. Interestingly, Huawei Watch 2, which uses a custom Wear OS, actually prefers WiFi

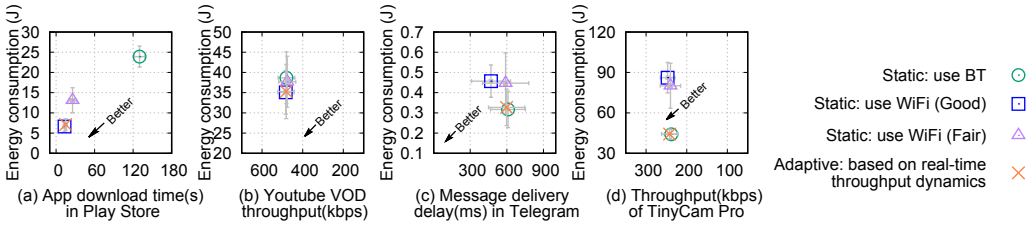


Fig. 12. QoE-energy tradeoffs across four real workloads using different interface selection policies (LG Urbane Watch, normal BT RSSI, Good/Fair WiFi network condition).

over BT, leading to potentially high energy consumption. In this section, we quantitatively analyze how the network selection policy affects the important tradeoff between performance and energy consumption, using real-world workload on COTS smartwatches.

5.1 Impact of Single-path Interface Selection

We first study the *single-path* interface selection, *i.e.,* using only one interface at any given time. We consider four real-world workloads: (a) downloading a wearable app of 16MB from the Google Play Store, (b) streaming a 2-min YouTube video to a watch, (c) delivering a short message by Telegram, and (d) streaming from an IP-camera in real time for 150s using the TinyCam app [5]. For these diverse workloads, we employ the app download time, the video throughput, the message delivery delay, and the real-time data streaming rate as the QoE metrics, respectively. Similar to that in Figure 8, we calculate the energy consumption using the energy model developed by [51]. Regarding the network selection policy, we consider the following four options: (1) always using BT, assuming a good network condition (-50 dBm RSSI), (2) always using WiFi, assuming a good network condition (10Mbps BW, 10ms RTT), (3) always using WiFi, assuming a fair network condition (5Mbps BW, 20ms RTT), and (4) an approach that dynamically switches between BT and WiFi as to be detailed in §5.2.

For each combination of the workload and network selection policy, we repeat the experiment 10 times. We show the results in Figure 12 to illustrate the tradeoff between QoE and energy consumption. Each plot in Figure 12 corresponds to a workload; each plot has four clusters corresponding to the four interface selection policies described above. Ideally, we prefer a cluster to be located in the bottom-left corner with a good QoE (the X Axis) while incurring a low energy overhead (the Y Axis). Our key observation from Figure 12 is that, depending on the app workload, the preferred interface selection policy differs. For (a) and (b), given their large data sizes, WiFi offers both lower energy consumption and a better QoE due to its higher throughput and higher energy efficiency (*i.e.,* joule per byte) compared to BT. In contrast, for (c), WiFi only marginally reduces the message delivery latency while incurring considerably higher energy consumption compared to BT. This is because the small message size and WiFi's high base power consumption lead to a higher joule per byte compared to BT. For (d), the workload consists of CBR traffic that BT can already sustain. This makes BT more energy-efficient than WiFi, which has a higher base power consumption and bandwidth under-utilization.

The energy results in Figure 12 only consider the energy consumed on the wearable. In addition, using BT incurs additional energy footprint on the paired smartphone that acts as a proxy forwarding traffic between the wearable and the server. The smartphone needs to utilize both its WiFi interface (with the server) and BT interface (with the wearable). To quantify such an energy overhead, we focus on the "static: use BT" scenario in Figure 12, and apply the smartphone WiFi [19] and BT [27] power models to calculate the overall smartphone-side radio energy consumption to be 121.984 J,

125.76 J, 0.524 J, and 157.2 J, respectively, for the four workloads. This non-trivial energy overhead on the smartphone makes it more complex to make interface selection decisions for the wearable.

The above results indicate that the static interface selection policy, which strictly prefers one interface over another as employed by almost all of today's smartwatches, does not always provide a preferred tradeoff between performance and energy consumption. The results suggest the need for a more adaptive interface selection policy. In §5.2, we will describe such an example corresponding to the "Adaptive" cluster in Figure 12.

5.2 Multipath Performance on Wearables

Multipath transport, which simultaneously uses multiple network interfaces, is becoming popular on smartphones, as fueled by standardized solutions such as MPTCP [1]. Despite a lack of prior work, we do believe that multipath transport can also benefit wearable networks in two aspects: (1) enhancing the throughput by aggregating bandwidth, and (2) facilitating seamless handover or fast interface switch. We examine the first aspect now and address the second one later.

We consider a common usage scenario involving a WiFi path and a BT path. In the wearable context, we do *not* expect multipath to be always used due to energy constraints. Instead, a wearable can *adaptively* enable multipath (*e.g.,* enhancing BT using WiFi) to meet user-specified deadlines or to prevent stalls for multimedia streaming [34]. Note that maintaining active WiFi connectivity incurs negligible power consumption due to WiFi's deep power-saving mode [13, 45, 66].

A Multipath Framework for Wearables. The Wear OS by default does not support multipath transport. Also, it is difficult to directly use MPTCP because BT does not speak TCP/IP by default. We thus make a methodological contribution of adding the multipath transport feature (over WiFi and BT) to the Wear OS. Specifically, we first leverage *ConnectivityManager* in the Wear OS to keep WiFi active when BT is also on. We then use the Linux socket API and Bluetooth API to build a custom multipath framework. In our framework, each path is a standalone TCP connection. The WiFi path is established directly between the wearable and the server[2], and the other path is wearable–*CPROXY*–server where the wearable–*CPROXY* segment is over BT. On the sender side, the original data stream is split into data chunks that are distributed onto the paths. We add to each chunk a custom header containing the metadata such as the size and global sequence number of the chunk. The receiver side then uses the metadata to reassemble the received chunks into the original data stream. To provide application transparency, we use `netfilter` [4] to transparently intercept application TCP connections on the wearable side. We also implement three off-the-shelf scheduler algorithms that determine how to distribute the traffic onto the paths: MPTCP's default minRTT scheduler [62], a round-robin scheduler, and a redundant scheduler. The first two schedulers help improve the throughput by aggregating the bandwidth of all paths; the third scheduler helps reduce the latency by sending duplicate data to all paths. Our system consists of around 10K lines of Java and C/C++ code. It is also open-sourced on GitHub [11].

Energy Overhead. We measure our multipath framework's energy overhead using a Monsoon power monitor [8]. Compared to the base power level of an LG Urbane Watch with the screen being turned off, our framework incurs only 0.6% of additional device-level power consumption. Some use cases such as fast interface switch further require our framework to keep the WiFi interface turned on and maintain a long-lived TCP subflow. We find that doing so incurs a device-level energy overhead of 6.2% based on an 8-hour measurement, using a 4-minute keep-alive timer as suggested by the RFC [15].

[2]In our experiments, to make our multipath framework fully transparent to the original server, we actually run the server-side code of our framework on an in-cloud proxy. The proxy–server path is verified not to be the performance bottleneck.

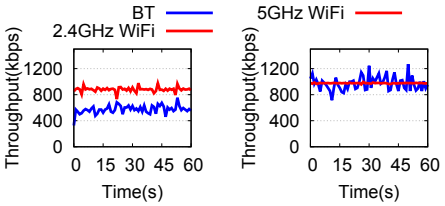


Fig. 13. BT-WiFi multipath under 2.4/5 GHz WiFi (Nexus 5, normal BT/WiFi RSSI).

Performance Aggregation Results on Wearables. Leveraging our wearable multipath framework, we conduct experiments on an LG Urbane paired with a Nexus 5X to assess the multipath performance over WiFi and BT. Other watch and phone pairs yield qualitatively similar performance. We focus on two types of improvements brought by multipath: the latency reduction when the redundant scheduler is used, and the bandwidth aggregation when the minRTT scheduler is used. For the latency reduction, we observe positive results. For example, using the redundant scheduler helps reduce the average RTT by 29% for CBR traffic at 500kbps (WiFi: 10Mbps BW, 10ms RTT; BT: -50 dBm RSSI). However, we find that the bandwidth aggregation results are much worse than our expectation. Ideally, for long-lived data transfers, the aggregated throughput achieved by multipath should be the sum of all paths' data rate. In reality, we observe that the bandwidth gain from multipath is far less than that. For example, when we throttle the WiFi and BT path's bandwidth to both 1Mbps, the multipath bandwidth gain compared to a single path is only 7%.

We realize the reason for the above unexpected results are multifold and cross-layer. For example, at the transport layer, we need a better scheduler that takes into account the heterogeneity between WiFi and BT. Very importantly, we also discover another key reason rooted deeply at the PHY layer: all our smartwatches support only 2.4 GHz WiFi that operates at the same frequency band of BT. The WiFi and BT thus cause interference when simultaneously transmitting data. This is confirmed by the following experiment: a Nexus 5 smartphone performs bulk data transfers over BT and WiFi at the same time (the phone supports both 2.4 GHz and 5 GHz WiFi), with the WiFi throughput being capped at 1Mbps. The left (right) plot in Figure 13 shows the BT and 2.4 GHz (5 GHz) WiFi throughput measured on the phone. As shown, compared to 5 GHz WiFi, when 2.4 GHz WiFi is used, the BT and WiFi throughput drops by 47% and 7%, respectively. Overall, our findings suggest the need for introducing 5 GHz WiFi on COTS wearables for reducing the WiFi-BT interference, in order to facilitate multipath transport over BT and WiFi.

Fast Interface Switch on Wearables. Recall from the beginning of this subsection that another important use case of multipath transport is to support fast interface switch, which seamlessly and transparently migrates a TCP connection from one path to another path without requiring re-establishing the connection. We utilize this feature to develop an adaptive interface selection policy corresponding to the "adaptive" cluster in Figure 12. Specifically, assuming an on-going download (the upload case is similar), our scheme uses BT over the *CPROXY* mode by default. Meanwhile, it monitors the number of bytes buffered at the *CPROXY* by tracking the incoming and outgoing bytes' to/from the *CPROXY*. When the buffer occupancy level exceeds B bytes for T_1 seconds (we empirically choose B=10KB and T_1=500ms), we switch to WiFi as BT does not drain the buffer fast enough. The switch from WiFi back to BT is triggered by a low WiFi throughput (we use $\leq$500kbps) for T_2 seconds (we use T_2=5 seconds).

We implement this adaptive interface selection strategy using our developed wearable multipath framework. The experimental results in Figure 12 suggest that it outperforms the static policies over all four workloads. In addition, we will apply multipath to improve the BT-WiFi handover performance in §6.

6 BT-WIFI HANDOVER PERFORMANCE

In previous sections, we consider the scenario where both the wearable and its paired phone are stationary. In reality, either device can be mobile. Consider a typical mobility scenario where a user wearing a smartwatch walks away from her paired smartphone placed on a table. As the user walks away, the wearable will lose its BT connectivity. In this case, ideally the wearable needs a seamless handover from BT to WiFi, an important feature that is missing on today's wearables as we will reveal in this section.

6.1 Wearable Handovers are Common

Although the theoretical BT range can be up to 100m [10], in real-world scenarios the range is much shorter due to attenuations incurred by obstructions or vendors' intentional reduction of the radio power for saving energy. For example, on the Samsung Galaxy Gear, the effective BT range is less than 2 meters based on our measurement. Due to such a short range, network handovers are likely to occur very frequently when a wearable moves away from the paired smartphone. To understand how often handovers occur "in the wild", we conduct an IRB-approved user study involving 10 voluntary users each wearing an LG Urbane Watch. The 10 participants consist of 4 students, 3 faculty members, and 3 staff members in a large U.S. university. 5 of them are female. We develop a data collector that infers handover events by monitoring the network interfaces' states (the method will be described in §6.2). The user study lasted for two months in 2017. During the daytime (9 AM – 9 PM) across all users, the median handover frequency is once every 1.6 hours. For some users, handovers can happen as frequently as every 7 minutes. The results suggest the need for properly handling handovers to provide smooth network switches.

6.2 Poor Wearable Handover Performance

Motivated by the user study, we quantify the handover performance on state-of-the-art wearables through controlled experiments.

Monitoring the Network State. A prerequisite for measuring handovers is to monitor the network state change. We capture the state of each network interface from the Wear OS's *ConnectivityManager* in the background. The state information includes whether the network interface is up, *i.e., available* or not, and whether the interface provides actual network connectivity, *i.e., connected* or not. For example, when a smartwatch is associating with the WiFi AP, its WiFi is available but not yet connected.

Experimental Setup. Our experiment focuses on understanding handovers from BT to WiFi (handovers from WiFi back to BT can be studied using similar methods). We keep both BT and WiFi enabled on the wearable (so both interfaces are available) and let the Wear OS use the default network management policy. We use two wearable apps to generate the traffic workload. The first is a simple app developed by us (conveniently called RTApp). It represents a typical wearable app developer's *best-effort* user-space implementation of the handover logic, which requires the synergy between both the app and the OS. Our app emulates the same traffic pattern as the tinyCam app (to be detailed soon), *i.e.,* downloading a data chunk of 3KB every 160ms from our server, generating 150kbps downlink traffic over TCP. When a handover occurs, the old interface (BT) will lose its connectivity and shortly after that, the connectivity will appear on the new interface (WiFi). At this time (detected through polling), our RTApp will establish a TCP connection over the new interface and resume the data transfer.

The second app we test for handover is the tinyCam security camera app [5]. It is a popular, professionally designed commercial app that requires continuous network connectivity to stream real-time video captured from an IP camera to a wearable. We perform a black-box test for this app

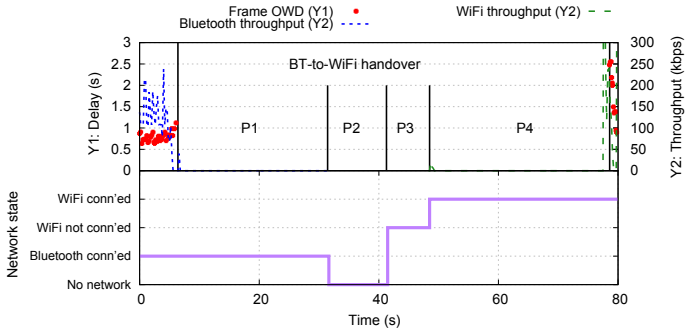


Fig. 14. Impact of BT-WiFi handover on QoE of tinyCam app (Huawei Watch, normal BT/WiFi RSSI).

to reveal its handover performance. We define two QoE metrics for the tinyCam app: (1) the *frame one-way delay (OWD)*, which is the time to transmit a video frame from the security camera to the watch (including the encoding and rendering time)[3], and (2) the *downlink throughput* on the watch.

Measurement Results. The top plot in Figure 14 shows the QoE metrics of the tinyCam app during a typical BT-WiFi handover: the frame OWD, the BT throughput, and the WiFi throughput. As shown, the app QoE severely degrades during the handover. At around t = 6s, the app stops receiving the video data from the SmartCam and the BT throughput drops to zero. The video transmission resumes over WiFi at around 72.5s, with high frame OWD observed at the beginning. We repeat this experiment for 10 times and the average "blackout" period during which the app does not receive any video data is surprisingly 70.0s. We then run the experiment under the same setting for our RTApp, whose average handover delay is measured to be 38.6s across 10 runs (we will explain the difference shortly). We further conduct the experiment on three different smartwatches and observe high handover delays on all of them as shown in Table 5. The results show that handovers are poorly handled by the Wear OS and/or the wearable app.

6.3 Root Cause of the High Handover Delay

To understand the root cause of the high handover delay, we break it down into four phases based on the captured network state information, as shown in the bottom plot in Figure 14: (P1) BT is still connected but the data cannot be actually transmitted due to poor signal strength, (P2) no network is available, (P3) the WiFi AP association period, *i.e.,* WiFi is available but not connected, and (P4) WiFi is connected but there is no application data transmission. The methodology for the breakdown analysis is as follows. For each network, our measurement tool (Table 3, §2) logs whether the network is ready to use by applications, *i.e.,* available or not, and whether the interface provides actual network connectivity, *i.e.,* connected or not, through Wear OS APIs. We then group both networks' logged states as shown in Figure 14.

Our analysis reveals two sources of delay that contribute to the overall handover latency: the delay from the Wear OS (P1, P2, and P3), as well as the delay incurred by the wearable app (P4). We next detail both types.

Delay from the Wear OS. Under the default network management policy of Wear OS based wearables, when BT is connected, WiFi is not available (*i.e.,* its interface is turned off by the OS) even if the device is under the coverage of both BT and WiFi. In this case, when the wearable moves away from the BT coverage, the Wear OS needs to: wait until the BT connectivity is completely

[3]To measure the frame OWD, we use a phone to display continuously increasing timestamps from a stopwatch app as the input stream to the SmartCam. The tinyCam app then shows the captured timestamp on the watch. The frame OWD can thus be calculated by comparing the timestamps when the same stopwatch frame appears on the phone and watch, whose timestamps are synchronized beforehand.

Table 5. BT-to-WiFi handover delay on 3 smartwatches for tinyCam app and RTApp (normal BT/WiFi RSSI).

	LG Urbane	LG Urbane 2nd	HUAWEI Watch
tinyCam	43.1 ± 5.7 s	52.9 ± 8.2 s	70.2 ± 9.7 s
RTApp	28.3 ± 2.6 s	14.3 ± 1.3 s	38.6 ± 5.3 s

lost as its RSSI drops below a threshold (P1), turn on the WiFi interface (P2), and then perform an AP association (P3). The whole process incurs a long period of time. Across the 10 runs on an LG Urbane watch, the average duration of P1, P2 and P3 are 12.9s, 15.5s and 8.3s, respectively, with their total duration accounting for 52% of the overall handover delay.

Delay Incurred by the Wearable App. We next investigate the wearable app's behavior during a handover event. In the tinyCam app case, even after WiFi gains its connectivity (after P3), the app still takes around 33.3s on average before the actual data transfer resumes over WiFi (P4). In contrast, our RTApp only takes 5.6s on average to resume the data transfer. Such a disparity of the P4 duration causes the two apps' vastly different handover duration shown in Table 5. In other words, although the handover completes after P3 from the OS's perspective, it takes additional time for the app to actually resume the data transfer (P4). The variation of P4 is very likely attributed to the app logic. Unfortunately, since Wear OS does not provide an API for seamlessly migrating data transfers between IP-based and non-IP networks, wearable apps need to implement their own data migration logic at the app layer. Doing so is tedious and challenging for average app developers.

6.4 Reducing the Handover Delay

We now design and implement a solution that reduces the handover delay. Our basic idea consists of the following. First, an important reason for Wear OS's bad handover performance is its *reactive* nature, *i.e.,* the WiFi connectivity is not established until the BT connectivity is fully torn down. Our scheme instead predicts a BT-WiFi handover by monitoring the BT channel quality. When the quality drops below a threshold (but BT is still usable), we *proactively* establish the WiFi connectivity and perform a handover to WiFi (assuming the WiFi channel quality is acceptable). Second, we leverage the multipath framework introduced in §5.2 to provide application transparency. Before a BT-WiFi handover, once the WiFi connectivity is established, the OS adds a new WiFi subflow to the corresponding TCP connection, and schedules future data to the WiFi subflow. No modification is needed at the user application, which always sees the same TCP connection. Third, our scheme further leverages *reinjection* to facilitate seamless data migration. Specifically, when it decides to perform a BT-WiFi handover, it also sends all unacknowledged (*i.e.,* "in-flight") data on the BT path, which may experience long delays due to its weak channel quality, to the WiFi path. In multipath transport, this is called packet reinjection, which trades a small number of redundant bytes for better performance (smoother handover in our case).

We implement the above design points and integrate them into our wearable multipath framework (§5.2). We use the BT RSSI as the BT channel quality metric [76], and empirically set its threshold for initiating a handover to -66dBm. A future research direction here is to further leverage the wearable's motion sensors or acoustic ranging [64] to precisely track the wearable's relative position to the phone in order to facilitate more accurate handover prediction. Using a similar approach, we also implement the handover mechanism from WiFi back to BT. We take two approaches to prevent oscillations between BT and WiFi. First, we use the Kalman filter to smooth the RSSI samples [18]. Second, after a BT-WiFi handover, we require the wearable to stay on WiFi for at least 5 seconds (a configurable parameter) unless the WiFi connectivity is lost.

Evaluation. We evaluate how our scheme helps accelerate the BT-WiFi handover process. The experimental setup is as follows. A user puts her Nexus 5X smartphone in a typical conference room (5m by 6m) and moves out of the room at a normal walking speed with a paired LG Urbane

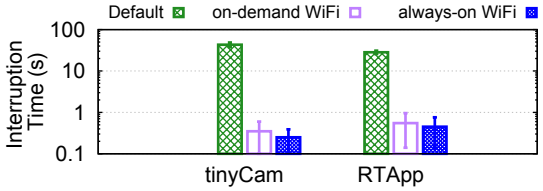


Fig. 15. Reducing the BT-to-WiFi handover delay for tinyCam/RTApp on an LG Urbane paired with Nexus 5X.

smartwatch worn on her wrist. As she walks out of the smartphone's BT coverage, the watch will experience a BT-WiFi handover. We use the two apps introduced in §6.2 as the workload: the tinyCam app that streams video contents from our camera in real-time, and our RTApp program that performs CBR streaming with improved application handover logic. The handover delay is measured using the same approach for generating Figure 14. We compare three handover schemes in Figure 15: "default" is the reactive handover approach used by Wear OS; "on-demand WiFi" corresponds to our proposed scheme where the WiFi connectivity and the multipath subflow are established in an on-demand fashion based on BT channel quality prediction; "always-on WiFi" also refers to our scheme, but we always maintain the WiFi connectivity and pre-establish the WiFi subflow to further speed up the handover. We repeat each experiment 10 times to overcome the randomness incurred by the user's walking paths. As shown in Figure 15, our scheme is extremely effective: it reduces the handover delay from more than 28s to less than 0.6s (123x and 51x reduction for tinyCam and RTApp, respectively). Note the Y axis is in log scale. For the "always-on WiFi" variation, the improvement is even higher (172x and 63x respectively) since the pre-established WiFi subflow allows immediate data transfers, but the cost is a slightly higher radio energy consumption (measured to be 6.2% as described in §5.2) compared to the "on-demand WiFi" variation.

7 RELATED WORK

Mobile Network Performance has been extensively studied in the past decade. Examples include crowd-sourced smartphone measurements [28, 39, 40], WiFi/LTE radio energy efficiency [38], network interface management [14, 65], power management [19, 68, 72], smartphone app performance [21, 57], and mobile multipath [56, 58]. Our work differs from the above by investigating wearable networking with unique characteristics. We propose novel measurement methodologies and gain new insights in the wearable context.

Wearable Systems. R. Liu *et al.* analyzed the execution of Android Wear OS and identified several inefficiencies [48, 49]. X. Liu *et al.* conducted a user study to understand smartwatch usage in the wild [51]. Kolamunna *et al.* studied the user behavior and application traffic characteristics for SIM-enabled wearables [44]. Chauhan *et al.* [17] characterized smartwatch apps. Hester *et al.* [35] developed an energy-efficient and multi-application wearable platform. There also exist studies on other aspects of wearable systems including display [54], storage [37], user interface [20, 78], energy [32, 79], and security [53, 77]. Researchers have also designed new sensing applications [60, 70] using wearables and studied the human-wearable interaction [69, 80]. Our work instead focuses on the networking aspect of commercial wearable OSes.

Improving Wireless Performance. There has been work on improving BT/BLE communications [24, 42, 47] and localization [36]. Compared to them, our study identified new BT performance issues on COTS wearables and provide guidelines for improving the wearable networking performance over BT. There have also been studies on bufferbloat [16, 50, 55, 63, 71]. We instead discovered severe and unique bufferbloat issues in the wearable context when a paired smartphone is serving as a proxy (§4.2). In addition, vertical handovers in wireless networks have also been

studied in the literature [43, 61, 67]. Here we focus on the handover between IP and non-IP networks, and consider both the OS and wearable application behaviors. We also develop a multipath framework to support bandwidth aggregation and seamless handover. It differs from existing work on mobile multipath [22, 25, 26, 30, 33, 34, 46, 56, 58, 59, 75] in that our framework focuses on WiFi/BT networks in the wearable context. We also identify unique performance issues that hinder wearable multipath on COTS wearables, such as the interference between BT and 2.4G Hz WiFi.

8 LIMITATIONS AND FUTURE WORK

We describe several limitations and future work of our study.

• Besides BT and WiFi, a wearable may have other network interfaces such as LTE and NFC. In this study, we focus on BT and WiFi due to their popularity and prevalence in today's wearable ecosystem (in particular, smartwatches [51, 79]). In our future work, we plan to study the interplay among more than two types of networks, such as BT, WiFi, LTE, and ZigBee, in terms of multipath, interface selection, and handover, in the wearable context.

• We only studied a limited number of real wearable apps (YouTube for wearable, TinyCam, Telegram Messenger, Play Store, *etc.*). This is mostly because most of today's wearable apps do not incur a significant amount of traffic. Nevertheless, we envision that future wearable apps will become more network-intensive as fueled by new hardware, OS support, and applications. Examples include continuous computer vision on smart glasses [23, 31], remote camera preview [6], real-time screen projection [2], and network-level collaboration between phone and watch [52]. We plan to study these new applications in our future work.

• Our study can also be extended to considering multiple devices that a person wears (*e.g.,* smartwatch, smart glasses, smart activity tracker, *etc.*). This creates novel use cases, but also brings in new challenges on, for example, energy-efficient content delivery in a body-net and choosing the appropriate Internet gateway when there are multiple such gateways.

• Finally, another limitation of our work is that all measurement results are based on the Wear OS, which, together with its predecessors (Android Wear), are expected to account for 41.8% of the market share of smartwatch OSes in 2020 [7]. There exist many other wearable OSes. In particular, due to its proprietary platform and closed ecosystem, it is difficult for us to perform an in-depth study of Apple's Watch OS. Nevertheless, we believe that the high-level lessons we learned from the Wear OS are general, and can benefit the design and implementation of future wearable OSes. Note that all our findings are applicable to old Android Wear 1.x versions as verified by us.

9 CONCLUDING REMARKS

We have learned many lessons from smartphones [38, 39] about the importance of properly handling the interaction between the lower-layer radio (*e.g.,* WiFi MAC and cellular RRC/RLC) and upper-layer protocols (*e.g.,* TCP, apps). Our study reveals that it is also the case for wearable networking. Indeed, many of its unique characteristics motivate us to conduct an in-depth investigation of the networking performance of Wear OS. We identify severe performance issues and make suggestions for improvements regarding several key aspects of wearable networking: Bluetooth performance, smartphone proxying, network selection, and handover. We believe our findings provide key knowledge and experiences for improving the networking subsystem of future wearable OSes.

ACKNOWLEDGEMENTS

We would like to thank our shepherd, Carey Williamson, and the anonymous reviewers for their valuable comments. This work is partially supported by NSF under the grants CCF-1628991 and CNS-1629763.

REFERENCES

[1] 2016. MPTCP v0.91 Release. http://multipath-tcp.org/pmwiki.php?n=Main.Release91.
[2] 2017. Cicret Bracelet. https://cicret.com/wordpress/.
[3] 2017. Telegram for Android Wear 2.0. https://telegram.org/blog/android-wear-2-0.
[4] 2017. The netfilter.org project. https://www.netfilter.org/.
[5] 2017. tinyCam Monitor PRO. https://play.google.com/store/apps/details?id=com.alexvas.dvr.pro.
[6] 2017. ZenWatch Remote Camera. https://play.google.com/store/apps/details?id=com.asus.rcamera2.
[7] 2018. Market share of smart wristwear shipments worldwide by operating system from 2015 to 2020. https://www.statista.com/statistics/466563/share-of-smart-wristwear-shipments-by-operating-system-worldwide/.
[8] 2018. Monsoon Power Monitor. https://www.msoon.com/online-store.
[9] 2018. Smartwatch Market Size, Share, Growth, Industry Report, 2018–2023. https://www.psmarketresearch.com/market-analysis/smartwatch-market.
[10] 2018. Specifications. The building blocks of all Bluetooth devices. https://www.bluetooth.com/specifications.
[11] 2019. *MPWear* github repository. https://github.com/XiaoShawnZhu/MPWear.
[12] 2019. *WearMan* github repository. https://github.com/XiaoShawnZhu/WearMan.
[13] Manish Anand, Edmund B Nightingale, and Jason Flinn. 2005. Self-tuning wireless network power management. *Wireless Networks* 11, 4 (2005), 451–469.
[14] Ganesh Ananthanarayanan, Venkata N Padmanabhan, Chandramohan A Thekkath, and Lenin Ravindranath. 2007. Collaborative downloading for multi-homed wireless devices. In *Mobile Computing Systems and Applications, 2007. HotMobile 2007. Eighth IEEE Workshop on*. IEEE, 79–84.
[15] Robert Braden. 1989. Requirements for Internet hosts-communication layers. (1989).
[16] Lawrence S. Brakmo and Larry L. Peterson. 1995. TCP Vegas: End to end congestion avoidance on a global Internet. *IEEE Journal on selected Areas in communications* 13, 8 (1995), 1465–1480.
[17] Jagmohan Chauhan, Suranga Seneviratne, Mohamed Ali Kaafar, Anirban Mahanti, and Aruna Seneviratne. 2016. Characterization of early smartwatch apps. In *PerCom Workshops*. IEEE.
[18] Dongyao Chen, Kang G Shin, Yurong Jiang, and Kyu-Han Kim. 2017. Locating and Tracking BLE Beacons with Smartphones. In *CoNEXT*. ACM.
[19] Xiaomeng Chen, Ning Ding, Abhilash Jindal, Y Charlie Hu, Maruti Gupta, and Rath Vannithamby. 2015. Smartphone energy drain in the wild: Analysis and implications. *ACM SIGMETRICS Performance Evaluation Review* 43, 1 (2015), 151–164.
[20] Xiang Chen, Tovi Grossman, Daniel Wigdor, and George Fitzmaurice. 2014. Duet: Exploring Joint Interactions on a Smart Phone and a Smart Watch. In *ACM CHI*.
[21] Xiaomeng Chen, Abhilash Jindal, Ning Ding, Yu Charlie Hu, Maruti Gupta, and Rath Vannithamby. 2015. Smartphone background activities in the wild: Origin, energy drain, and optimization. In *MobiCom*. ACM.
[22] Yung-Chih Chen, Yeon-Sup Lim, Richard J. Gibbens, Erich M. Nahum, Ramin Khalili, and Don Towsley. 2013. A Measurement-based Study of MultiPath TCP Performance over Wireless Networks. In *IMC*.
[23] Zhuo Chen, Lu Jiang, Wenlu Hu, Kiryong Ha, Brandon Amos, Padmanabhan Pillai, Alex Hauptmann, and Mahadev Satyanarayanan. 2015. Early implementation experience with wearable cognitive assistance applications. In *WearSys workshop*. ACM, 33–38.
[24] Zicheng Chi, Yan Li, Hongyu Sun, Yao Yao, Zheng Lu, and Ting Zhu. 2016. B2W2: N-Way Concurrent Communication for IoT Devices. In *SenSys*. ACM.
[25] Quentin De Coninck, Matthieu Baerts, Benjamin Hesmans, and Olivier Bonaventure. 2016. A First Analysis of Multipath TCP on Smartphones. In *17th International Passive and Active Measurements Conference*, Vol. 17. Springer.
[26] Shuo Deng, Ravi Netravali, Anirudh Sivaraman, and Hari Balakrishnan. 2014. WiFi, LTE, or Both? Measuring Multi-homed Wireless Internet Performance. In *IMC*.
[27] Roy Friedman, Alex Kogan, and Yevgeny Krivolapov. 2013. On power and throughput tradeoffs of wifi and bluetooth in smartphones. *IEEE Transactions on Mobile Computing* 12, 7 (2013), 1363–1376.
[28] Kensuke Fukuda, Hirochika Asai, and Kenichi Nagami. 2015. Tracking the evolution and diversity in network usage of smartphones. In *IMC*. ACM.
[29] Yihua Guo, Feng Qian, Qi Alfred Chen, Zhuoqing Morley Mao, and Subhabrata Sen. 2016. Understanding On-device Bufferbloat for Cellular Upload. In *IMC*. ACM.
[30] Yihua Ethan Guo, Ashkan Nikravesh, Z Morley Mao, Feng Qian, and Subhabrata Sen. 2017. Accelerating multipath transport through balanced subflow completion. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*. ACM, 141–153.
[31] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2014. Towards wearable cognitive assistance. In *MobiSys*. ACM.

[32] MyungJoo Ham, Inki Dae, and Chanwoo Choi. 2015. LPD: Low Power Display Mechanism for Mobile and Wearable Devices.. In *USENIX ATC*.
[33] Bo Han, Feng Qian, Shuai Hao, and Lusheng Ji. 2015. An Anatomy of Mobile Web Performance over Multipath TCP. In *CoNEXT*.
[34] Bo Han, Feng Qian, Lusheng Ji, and Vijay Gopalakrishnan. 2016. MP-DASH: Adaptive video streaming over preference-aware multipath. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM, 129–143.
[35] Josiah Hester, Travis Peters, Tianlong Yun, Ronald Peterson, Joseph Skinner, Bhargav Golla, Kevin Storer, Steven Hearndon, Kevin Freeman, Sarah Lord, et al. 2016. Amulet: An Energy-Efficient, Multi-Application Wearable Platform. In *SenSys*. ACM.
[36] AKM Mahtab Hossain and Wee-Seng Soh. 2007. A comprehensive study of bluetooth signal parameters for localization. In *PIMRC*. IEEE.
[37] Jian Huang, Anirudh Badam, Ranveer Chandra, and Edmund B. Nightingale. 2015. WearDrive: Fast and Energy-Efficient Storage for Wearables. In *USENIX ATC*.
[38] Junxian Huang, Feng Qian, Alexandre Gerber, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. 2012. A close examination of performance and power characteristics of 4G LTE networks. In *MobiSys*. ACM.
[39] Junxian Huang, Feng Qian, Yihua Guo, Yuanyuan Zhou, Qiang Xu, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. 2013. An in-depth study of LTE: effect of network protocol and application behavior on performance. In *SIGCOMM*. ACM.
[40] Junxian Huang, Qiang Xu, Birjodh Tiwana, Z Morley Mao, Ming Zhang, and Paramvir Bahl. 2010. Anatomizing application performance differences on smartphones. In *MobiSys*. ACM.
[41] Haiqing Jiang, Yaogong Wang, Kyunghan Lee, and Injong Rhee. 2012. Tackling Bufferbloat in 3G/4G Networks. In *IMC*. ACM.
[42] Aditya Karnik and Anurag Kumar. 2000. Performance analysis of the Bluetooth physical layer. In *Personal Wireless Communications*. IEEE.
[43] Kyu-Han Kim, Yujie Zhu, Raghupathy Sivakumar, and Hung-Yun Hsieh. 2005. A receiver-centric transport protocol for mobile hosts with heterogeneous wireless interfaces. *Wireless Networks* 11, 4 (2005), 363–382.
[44] Harini Kolamunna, Ilias Leontiadis, Diego Perino, Suranga Seneviratne, Kanchana Thilakarathna, and Aruna Seneviratne. 2018. A First Look at SIM-Enabled Wearables in the Wild. In *IMC*. ACM.
[45] Ronny Krashinsky and Hari Balakrishnan. 2002. Minimizing energy for wireless web access with bounded slowdown. In *Proceedings of the 8th annual international conference on Mobile computing and networking*. ACM, 119–130.
[46] HyunJong Lee, Jason Flinn, and Basavaraj Tonshal. 2018. RAVEN: Improving Interactive Latency for the Connected Car. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*. ACM, 557–572.
[47] Amit A Levy, James Hong, Laurynas Riliskis, Philip Levis, and Keith Winstein. 2016. Beetle: Flexible communication for bluetooth low energy. In *MobiSys*. ACM.
[48] Renju Liu, Lintong Jiang, Ningzhe Jiang, and Felix Xiaozhu Lin. 2015. Anatomizing System Activities on Interactive Wearable Devices. In *APSys*.
[49] Renju Liu and Felix Xiaozhu Lin. 2016. Understanding the Characteristics of Android Wear OS. In *MobiSys*. ACM.
[50] Shao Liu, Tamer Başar, and Ravi Srikant. 2008. TCP-Illinois: A loss-and delay-based congestion control algorithm for high-speed networks. *Performance Evaluation* 65, 6-7 (2008), 417–440.
[51] Xing Liu, Tianyu Chen, Feng Qian, Zhixiu Guo, Felix Xiaozhu Lin, Xiaofeng Wang, and Kai . Chen. 2017. Characterizing Smartwatch Usage in the Wild. In *MobiSys*. ACM.
[52] Xing Liu, Yunsheng Yao, and Feng Qian. 2017. Rethink Phone-Wearable Collaboration From the Networking Perspective. In *ACM WearSys*.
[53] Xiangyu Liu, Zhe Zhou, Wenrui Diao, Zhou Li, and Kehuan Zhang. 2015. When good becomes evil: Keystroke inference with smartwatch. In *CCS*. ACM.
[54] Hongyu Miao and Felix Xiaozhu Lin. 2016. Tell Your Graphics Stack That the Display Is Circular. In *HotMobile*.
[55] Kathleen Nichols and Van Jacobson. 2012. Controlling queue delay. *Commun. ACM* 55, 7 (2012), 42–50.
[56] Ana Nika, Yibo Zhu, Ning Ding, Abhilash Jindal, Y Charlie Hu, Xia Zhou, Ben Y Zhao, and Haitao Zheng. 2015. Energy and performance of smartphone radio bundling in outdoor environments. In *WWW*. ACM.
[57] Ashkan Nikravesh, Qi Alfred Chen, Scott Haseley, Xiao Zhu, Geoffrey Challen, and Z Morley Mao. 2018. QoE Inference and Improvement Without End-Host Control. In *SEC*. IEEE.
[58] Ashkan Nikravesh, Yihua Guo, Feng Qian, Z Morley Mao, and Subhabrata Sen. 2016. An in-depth understanding of multipath TCP on mobile devices: measurement and system design. In *MobiCom*. ACM.
[59] Ashkan Nikravesh, Yihua Guo, Xiao Zhu, Feng Qian, and Z Morley Mao. 2019. MP-H2: A Client-only Multipath Solution for HTTP/2. In *MobiCom*. ACM.

[60] Shahriar Nirjon, Jeremy Gummeson, Dan Gelb, and Kyu-Han Kim. 2015. Typingring: A wearable ring platform for text input. In *MobiSys*. ACM.
[61] Christoph Paasch, Gregory Detal, Fabien Duchene, Costin Raiciu, and Olivier Bonaventure. 2012. Exploring mobile/WiFi handover with multipath TCP. In *Proceedings of the 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design*. ACM, 31–36.
[62] Christoph Paasch, Simone Ferlin, Ozgu Alay, and Olivier Bonaventure. 2014. Experimental Evaluation of Multipath TCP Schedulers. In *ACM SIGCOMM Capacity Sharing Workshop (CSWS)*. ACM.
[63] Rong Pan, Preethi Natarajan, Chiara Piglione, Mythili Suryanarayana Prabhu, Vijay Subramanian, Fred Baker, and Bill VerSteeg. 2013. PIE: A lightweight control scheme to address the bufferbloat problem. In *High Performance Switching and Routing (HPSR), 2013 IEEE 14th International Conference on*. IEEE, 148–155.
[64] Chunyi Peng, Guobin Shen, and Yongguang Zhang. 2012. BeepBeep: A high-accuracy acoustic-based system for ranging and localization using COTS devices. *ACM Transactions on Embedded Computing Systems* 11, 1 (2012), 4.
[65] Trevor Pering, Yuvraj Agarwal, Rajesh Gupta, and Roy Want. 2006. Coolspots: reducing the power consumption of wireless mobile devices with multiple radio interfaces. In *MobiSys*. ACM.
[66] Daji Qiao and Kang G Shin. 2005. Smart power-saving mode for IEEE 802.11 wireless LANs. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, Vol. 3. IEEE, 1573–1583.
[67] Ahmad Rahmati, Clay Shepard, Chad Tossell, Angela Nicoara, Lin Zhong, Phil Kortum, and Jatinder Singh. 2010. Seamless flow migration on smartphones without network support. *arXiv preprint arXiv:1012.3071* (2010).
[68] Marcel-Catalin Rosu, C Michael Olsen, Chandrasekhar Narayanaswami, and Lu Luo. 2004. Pawp: A power aware web proxy for wireless lan clients. In *Mobile Computing Systems and Applications, 2004. WMCSA 2004. Sixth IEEE Workshop on*. IEEE, 206–215.
[69] Matthias Seuter, Max Pfeiffer, Gernot Bauer, Karen Zentgraf, and Christian Kray. 2017. Running with Technology: Evaluating the Impact of Interacting with Wearable Devices on Running Movement. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 1, 3 (2017), 101.
[70] Sheng Shen, He Wang, and Romit Roy Choudhury. 2016. I am a Smartwatch and I can Track my User's Arm. In *MobiSys*. ACM.
[71] Dan Siemon. 2013. Queueing in the Linux network stack. *Linux Journal* 2013, 231 (2013), 2.
[72] Jacob Sorber, Nilanjan Banerjee, Mark D Corner, and Sami Rollins. 2005. Turducken: hierarchical power management for mobile devices. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*. ACM, 261–274.
[73] Peng Sun, Minlan Yu, Michael J Freedman, and Jennifer Rexford. 2011. Identifying performance bottlenecks in CDNs through TCP-level monitoring. In *Proceedings of the first ACM SIGCOMM workshop on Measurements up the stack*. ACM, 49–54.
[74] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescape. 2011. Broadband Internet Performance: A View From the Gateway . In *ACM SIGCOMM*.
[75] Yeon sup Lim, Yung-Chih Chen, Erich M. Nahum, Don Towsley, Richard J. Gibbens, and Emmanuel Cecchet. 2015. Design, Implementation and Evaluation of Energy-Aware Multi-Path TCP. In *CoNEXT*.
[76] David Tse and Pramod Viswanath. 2005. *Fundamentals of wireless communication*. Cambridge university press.
[77] He Wang, Ted Tsung-Te Lai, and Romit Roy Choudhury. 2015. Mole: Motion leaks through smartwatch sensors. In *MobiCom*. ACM.
[78] Jian Xu, Qingqing Cao, Aditya Prakash, Aruna Balasubramanian, and Donald E. Porter. 2017. UIWear: Easily Adapting User Interfaces for Wearable Devices. In *ACM MobiCom*.
[79] Yi Yang and Guohong Cao. 2017. Characterizing and optimizing background data transfers on smartwatches. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. IEEE, 1–10.
[80] Cheng Zhang, Xiaoxuan Wang, Anandghan Waghmare, Sumeet Jain, Thomas Ploetz, Omer T Inan, Thad E Starner, and Gregory D Abowd. 2017. FingOrbits: interaction with wearables using synchronized thumb movements. In *Proceedings of the 2017 ACM International Symposium on Wearable Computers*. ACM, 62–65.

Received December 2018; revised January 2019; accepted February 2019