meSDN: Mobile Extension of SDN

Jeongkeun Lee,[1] Mostafa Uddin,[2] Jean Tourrilhes,[1] Souvik Sen,[1] Sujata Banerjee,[1]
Manfred Arndt,[3] Kyu-Han Kim,[1] Tamer Nadeem[2]

[1]HP Labs, [2]Old Dominion University, [3]HP Networking

ABSTRACT

Mobile devices interact wirelessly with a growing proliferation of cloud-based applications. Due to significant traffic growth and a wide variety of multimedia solutions, enterprise IT departments are demanding more fine-grained visibility and control of mobile traffic. They want to deliver optimal performance and a high quality of experience to a variety of users and applications. In the wired world, Software-Defined Networking (SDN) is a technology being embraced to deliver performance guarantees to end users by dynamically orchestrating quality of service (QoS) policies on edge switches and routers. Guaranteeing performance in a wired access network does not require any network control on clients, because the last hop between the network edge and wired device is a dedicated point-to-point link (e.g. Ethernet). However, this is not the case with wireless LANs (WLAN), since the last hop is a shared half-duplex medium and the WiFi MAC protocol does not allow access points to coordinate client uplink transmissions or 802.11 QoS settings. Hence, we argue that the SDN paradigm needs to be extended to mobile clients to provide optimal network performance between the cloud and wirelessly-connected clients. In this paper, we propose a framework called *meSDN* and demonstrate that it enables WLAN virtualization, application-aware QoS and improves power-efficiency from our prototype on Android phones.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Centralized networks*

Keywords

SDN, OpenFlow, WLAN, mobile

1. INTRODUCTION

Recent mobile cloud applications require guaranteed network performance more than conventional client-server applications as such mobile applications incur tight and real-time interactions with cloud and sometimes offload network-intensive workloads to cloud [1,2]. In a wired infrastructure, Software-Defined Networking (SDN) APIs is used to guarantee performance by dynamically coordinating network edges, mostly Ethernet switches. Recent SDN controllers also interact directly with cloud-based application servers to communicate application specific requirements [3]. This SDN(network)↔cloud(server) interaction, together with the existing mobile(client)↔cloud(server) interaction, aims to deliver application-optimized network performance and eventually bottleneck-free computing experience to users.

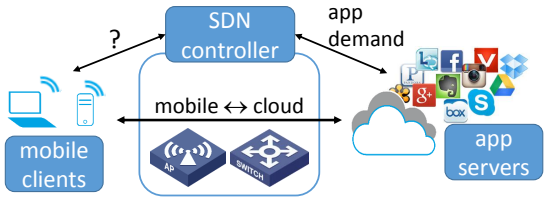


Figure 1: Mobile-Network-Cloud interactions.

Mobile(client)↔SDN(network) interaction is very essential to complete the loop between mobile, SDN and cloud (Figure 1). Knowing the application requirement from the cloud(server), it is important for the SDN(network) to provide performance guarantee to the client device. Unfortunately, the control of the existing SDN frameworks stops at the network edge, either at enterprise edge switches [4], datacenter hypervisor virtual switches [5] or home routers [6]. In wired environments, SDN policy can be effectively enforced on the traffic to/from end devices without modifying the end devices since the last hop is a point-to-point full-duplex link and transmissions from end devices do not interfere with each other. However, this is not the case with wireless LANs (WLAN), since the last hop is a shared half-duplex medium and the WiFi MAC protocol does not allow access points (APs) to coordinate client uplink transmissions.

SDN-enabled APs and infrastructure, such as OpenRadio and OpenRAN [7, 8], can control wireless resource usages for AP-to-client downlink transmissions; but they cannot control uplink traffic, which also interferes with downlink transmissions due to the half-duplex MAC of WiFi. Thus, AP-side control only cannot *guarantee* wireless resource for both uplink and downlink. Without resource guarantees, the performance experienced by end users can be highly unpredictable even with high transmission rates of recent 802.11n/ac.

MCS'14, June 16, 2014, Bretton Woods, New Hampshire, USA.

http://dx.doi.org/10.1145/2609908.2609948.

In addition to uplink transmissions, we should also manage uplink 802.11 QoS specification. For example, one client greedily setting its every uplink transmission as high priority can unfairly dominate the air-time resource; client-side QoS control via SDN APIs can prevent such unfair resource usage across network slices and also within the same slice. Because it is a shared medium, the wireless link is often the most critical from an end-to-end QoS perspective and most likely to define the performance perceived by the application and user. By integrating SDN APIs in the client device, we can manage uplink QoS over the shared wireless medium, and provide truly end-to-end QoS control.

In this paper, we argue that extending SDN control to end devices can support client↔network interaction capabilities and services such as guaranteeing airtime resource and E2E QoS for mobile clients. We propose a framework, *meSDN*[1] (mobile extension of SDN), that utilizes Open vSwitch [5] in monitoring and managing mobile's application traffic. *meSDN* also provides application-awareness. To set and enforce SDN policies correctly, we need accurate, fine-grained and trusted information of the user applications generating network traffic. Since *meSDN* resides on the client devices, it easily obtains the client application information and uses it for pTDMA scheduling and QoS control.

As a proof-of-concept, we use *meSDN* framework to design, implement and evaluate a WLAN virtualization service that slices end devices using a Time Division Multiple Access (TDMA) like airtime scheduling, named pseudo TDMA (pTDMA), that runs on top of 802.11 MAC. By using a modified Linux Qdisc on end devices, pTDMA virtualizes (separates) airtime resource between network slices while minimizing contention between clients within a slice. pTDMA also allows client WiFi interfaces to more efficiently utilize their active time and to sleep longer outside of the given transmission windows. We will present *meSDN*'s network virtualization capability and its improved power-efficiency from our prototyping on Android phones.

2. THE CASE FOR meSDN

Why SDN at mobile endpoints? In *meSDN*, we bring the client device's apps and their network usage into the SDN framework by running an agent in the client device, which is controlled from the network infrastructure. This concept of centrally managed agent software monitoring and controlling client devices is starting to become mainstream due to the Bring-Your-Own-Device (BYOD) phenomenon in enterprise networks. Enterprises have been deploying management software on their employee devices including personal mobile devices, to ensure device health, security and to protect corporate data stored in the devices [9]. Virtual Private Network (VPN) client software often perform bandwidth and access control on end devices. Mobile WAN optimization solutions use client-side software that interacts with a network proxy to optimize end-to-end performance [10].

However, to our best knowledge, none of them brought the client software into the SDN framework or tried direct coordination between the client agent and the network infrastructure (e.g., WLAN APs) to enable an integrated E2E solution. *We believe that running special-purpose client software is akin to extending opaque network middleboxes into end-devices and further fragmenting and complicating the control and management planes.* Instead, a better option would be to extend existing open and powerful SDN APIs and the associated control framework to the client side to support enhanced security, QoS and WLAN virtualization.

With the SDN APIs in clients, *meSDN* can enforce SDN polices directly on the client uplink traffic, for example, Call Admission Control (CAC) for resource management and Network Access Control (NAC) for security. CAC and NAC can be enforced at network edges but cannot prevent the controller traffic from consuming wireless resources. *meSDN* can avoid such wasted transmissions and can benefit all WLAN users. *meSDN* can also better utilize multiple wireless interfaces on a multi-homed mobile device [11].

Mobile devices today are capable of fulfilling the role of extended SDN control since smartphones, tablets and laptops are becoming more powerful, as opposed to CPU-limited wireless APs and Ethernet switches.

WLAN Virtualization is becoming a key to enable a more effective sharing of wireless resources by a diverse set of users with diverse requirements. For example, mobile carriers want to obtain a guaranteed share of RF resources on public WiFi infrastructure (e.g., in an airport) to offload their subscribers' data traffic to WiFi. Enterprises/Homes want to virtualize their WLAN infrastructure to create differentiated service networks, e.g., an employee/parents network vs. a guest/kids network. This kind of virtualization is already happening in cellular networks, e.g., Mobile Virtual Network Operators [12], and there is a clear need to extend this capability to WLANs.

The existing SDN mechanism for virtualizing the wired Ethernet infrastructure is not directly applicable to virtualize the WLANs. The last WLAN hop is a shared wireless medium and most WiFi clients today support only contention-based MAC, leaving no means for APs, even OpenFlow-enabled WiFi APs, to control client-to-AP uplink transmissions. Consider an example of an enterprise wanting to guarantee 50% share of airtime for employee devices. If there are nine guest devices and only one employee device, all with heavy backlog traffic to send, the employee will get only about 10% share of air-time. Previous approaches for APs to control clients' uplink 802.11 or TCP transmissions were either intrusive (dropping uplink 802.11 frames or downlink TCP ACKs), unfair by causing starvation (greedy use of Self-CTS) or hard to be adopted (requiring changes of 802.11 protocol) [13–15]. Furthermore, most of these techniques don't provide a good resource guarantee, since they only indirectly control uplink transmission probabilities and cannot prevent aggressive UDP streaming from monopolizing a link.

In summary, *meSDN* is motivated by the unique difference of wireless (vs. wired) and aims to provide fundamental software-defined solutions for many problems such as WLAN virtualization, application-awareness, E2E QoS and network troubleshooting. To realize them, *meSDN requires more than a trivial extension of existing SDN APIs* and we will discuss the proposed architecture in the following section.

In this paper, we primarily focus on enterprise settings where the WLAN infrastructure is managed by a single operator. Enterprise WLANs today implement Radio Resource Management (RRM) solutions [16–18] in a central controller (onsite or in the cloud), which collects channel, interference

[1]Pronounced as '*mee*-SDN'

and traffic information from the APs and controls channel, transmit power and user loads across APs, similar to the centralized solutions studied in the literature [19–21]. *meSDN* can leverage the centralized RRM solutions to efficiently achieve WLAN virtualization. Enterprise ITs often deploy device management software on employee devices [9], making client-side changes for *meSDN* easier. Later, we will discuss applying *meSDN* to non-enterprise settings.

3. meSDN

3.1 Architecture

As shown in Fig. 2, *meSDN* has three components in mobile clients: (1) Scheduler (e.g., Linux qdisc) (2) flow manager (e.g., Open vSwitch, OVS), and (3) local controller. There is also a global network controller that talks with the client local controllers and the APs. We assume SDN-enabled APs, which optionally provide a few useful details, e.g., beacon schedule and per-client airtime usage, to the global controller.

Flow Manager is a software OpenFlow switch, e.g. OVS, that measures per-flow statistics including our additional metrics, such as burst duration & rate and inter-burst time, and feeds them to the local controller for airtime scheduling, for example, in order to minimize the scheduling latency experienced by realtime applications. OVS also takes per-flow QoS and access control actions: it ensures correct QoS markings (IP DSCP/TOS) for end-to-end QoS provisioning and correct mapping to 802.11 QoS queues, (e.g. ensure that no P2P traffic gets queued in the Voice queue). In addition, *meSDN* extends OVS further to interact with WiFi driver to configure, for example, its power-save settings and also to collect 'per-flow' wireless statistics such as RSSI and drop count; this "Wireless Extension" enables the control plane to better schedule airtime resource. For example, VoIP flow suffering hidden-interference will show high drop counts and we can schedule the flow or client in a different time window to avoid interference. The 'per-flow' stat is useful because we can prioritize a VoIP flow over a p2p flow when both suffer high wireless drops.

Scheduler is Linux Qdisc that applies prioritization and rate-limiting to outgoing flows. OVS implementation today already leverages Qdisc to implement OpenFlow QoS APIs – prioritization and rate-limiting. To implement airtime scheduling, which is specific to wireless, *meSDN* can extend either Qdisc or WiFi driver. (We will compare the two options later.) The "*meSDN* extension" module, either in Qdisc or WiFi driver, starts/stops dequeueing of the outgoing flow based on the airtime schedule given by the control plane. Ideally, the ability to schedule airtime usage of 'each flow' is desirable for finer-grained airtime/QoS control and we can further extend OpenFlow APIs to signal per-flow schedules down to the "*meSDN* extension" module. But this would complicate the controller and scheduler implementations. Though we are open to the possibility of per-flow airtime scheduling, at this point, we assume airtime scheduling is done per-device and don't consider OpenFlow as a signaling API.

Local controller is userspace software that controls Flow Manager and Scheduler. The local controller also provides application-awareness and generates *flow-to-application mappings* by monitoring active network sockets (`netstat` logs) and their `pid/uid` bindings that are available in Android

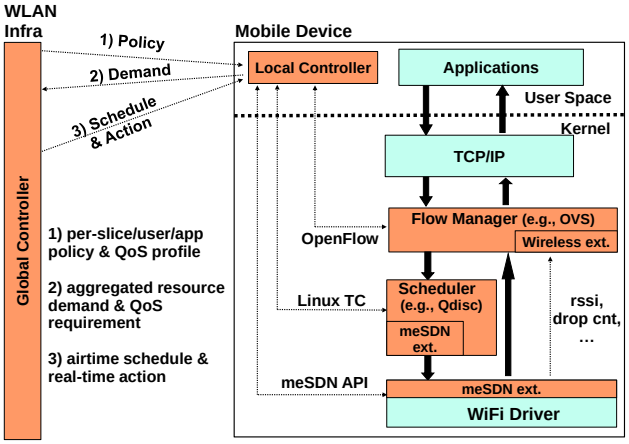


Figure 2: *meSDN* Architecture.

and other major operating systems. The local controller inserts a flow rule corresponding to each socket into OVS. Because the local controller knows which socket/flow belongs to which application, it can easily apply appropriate application-specific policy, assigned by the global controller or set directly by the end user [22]. We extend the OpenFlow APIs to read per-flow wireless stats from the flow manager. The local controller can use Linux Traffic Control (TC) or *meSDN* API to control the *meSDN* extension depending on it's location either in Qdisc or WiFi driver.

Global controller coordinates with the local controller in three steps (also see Fig. 2). First, the global controller provides per-slice, per-user, per-application policies and QoS profiles to the local controllers. For example, an enterprise IT may allow a certain VoIP application for guest users but not for employees. The IT admins may have pre-profiled a QoS spec for certain video applications. Second, given the policies and QoS profiles, the local controller can "aggregate" currently running applications' resource (e.g., airtime) and QoS requirements (e.g., latency tolerance), and send the aggregated requirements to the global controller scheduler (the 2nd step in Fig. 2). This client-side aggregation reduces control overhead and improves scalability. This also protects user privacy because the local controller can provide network requirements to the global controller without revealing which applications are running on the device. This is in contrast to current network-based application detection solutions that employ Deep Packet Inspection (DPI), looking into the user payload.

Finally, the global controller apply proper actions back to the local controllers (the 3rd step). For example, based on the received per-client aggregate airtime demands, the global controller provides the schedules to the local controller, which then program scheduling mechanism through *meSDN* extension.

The global controller may also directly manage the device components (OVS and qdisc) using OpenFlow, and since OVS supports multiple controllers, it can be managed by both local and global controllers. Unlike wired SDNs that typically have an out-of-band control channel between switches and the controller, we have to use the same wireless interface for data and control; and reliable and scalable in-band control is hard in wireless due to power-saving and

interference. Thus, we argue that *the global controller is better to communicate only with the local controller, which runs as a proxy for the device components*, in semi-real-time synchronously e.g., once every 'N' beacon cycles, and also asynchronously as needed, e.g., to adapt to sudden changes in application demands. We leave the control message scheduling in *meSDN* as future work.

3.2 meSDN applications

meSDN enables the control-plane of wireless networks to be extended to mobile devices and allows for top-level decisions to be made from a global network controller with knowledge of the network as a whole, rather than device-centric configurations. In addition, *meSDN* easily obtains user application information, as well as the ability to monitor and control application flows dynamically. We demonstrate several use-cases of *meSDN* as follows.

App-aware E2E QoS is one of key SDN applications. Existing approaches require user inputs or custom integration between a SDN controller and each application to detect the start/end of application flows and learn about their QoS requirements [3, 4, 23]. Deep-Packet Inspection and machine learning approaches leverage unique per-app signatures and distinguish network flows from different applications [24]; but they are unable to differentiate various types of network flows generated from one application. For example, Skype generates voice, video, screen sharing, IM, file transfer, and signaling messages, each with different requirements and constraints. Realtime detection of QoS demanding flows would require constant monitoring and analysis of network flow patterns (packet sizes, inter-packet time), which is costly to implement on CPU-limited APs and switches.

In our prototype system, we could collect per-flow packet size and timestamp information from Android devices with marginal overhead by extending OVS and OpenFlow stat APIs; we could detect Skype video flows and correctly set their QoS to Video class using OVS/OpenFlow. *meSDN* can also notify the global controller of the start of the video flow and ask to enforce a consistent QoS policy across the entire E2E flow path. Because the controller has global visibility beyond that single link, it can detect a bottleneck and do an admission control of the video flow from the client. E2E QoS coordination is our future work. Learning per-app/flow transmission pattern will be discussed in the next section.

Network fault diagnosis and trouble shooting continues to plague many users today – so, having the ability to truly inspect flows end-to-end and conduct diagnostic tests at the endpoints may be one of the killer applications of *meSDN*.

As an example, after installing OVS in the version of Android we tested, the test phone failed to connect to an AP. By analyzing OVS logs and flow table dump with our timestamp extension, we could find the last packet went through the OVS datapath was an 802.1x packet (eth_type: 0x888e) received from the WiFi interface `wlan0` and sent to the internal bridge interface (`br0`); we could scope down the problem – `br0` failed to forward the 802.1x packet to *wpa_supplicant* application that is in charge of WiFi authentications. To automate this process, one can write an fault-diagnosis app in the *meSDN* local controller that dumps and analyzes flow tables, for example when WiFi connection establishment takes more than X seconds.

WLAN virtualization with guaranteed airtime slicing is the main application of *meSDN*, which we will detail in the next section of pTDMA.

3G/4G cellular network radio management policies are known to badly interact with mobile applications that employ periodic transfers, wasting radio resources and device energy [25,26]. The periodic transfers are typically for background analytics and delay-tolerant, thus can wait to be sent together with user-triggered data transfers. *meSDN* framework and pTDMA airtime scheduler can be used to detect and delay such periodic transfers without expecting the source applications to optimize their transmission patterns.

4. pTDMA

As a proof-of-concept of *meSDN* framework, we design, prototype and evaluate a WLAN virtualization service, pTDMA, while leaving its full design & implementation to future work.

4.1 Overview

pTDMA, using TDMA-like 'coarse-grained' airtime scheduling on top of 802.11 CSMA/CA MAC, virtualizes airtime resource between network slices. Note that we do not (and cannot) guarantee absolute interference-free airtime, which is impossible in unlicensed bands. pTDMA guarantees a share of airtime duration for each slice to 'attempt' medium access while avoiding contention between different slices and minimizing contention between clients within a slice; but it cannot completely avoid interference.

We assume the wireless "channel" resource is under control by the centralized Radio Resource Management (RRM) solutions in enterprise WLANs [16–18]. We define the role of pTDMA to manage airtime share between virtual network instances (their clients) that collocate in space and channel. RRM can compute airtime available for each channel and AP based on the measured contention and interference and feed the airtime availability to our global pTDMA scheduler.

The key concept of pTDMA is to *separate airtime slices used by different network instances* so that traffic for different networks do not overlap and wireless contention is confined to traffic internal to one network instance. Fig. 3 illustrates a simplified example of the airtime resource shared by two network instances: employee network and guest network with 50:50 time share. As a simple baseline, the entire time slice (max 50%) can be open to all clients of each network instance. The clients and APs of each network instance will contend for the medium access based on the 802.11 MAC. If the operator also wants to manage contention and airtime usage within each instance, the instance's airtime slice can be further divided into multiple time windows and scheduled based on each individual clients' QoS requirements and traffic pattern provided by the local controllers.

4.2 Scheduling principles

There are myriads of scheduling schemes in the literature that addressed other important factors like work-conservation, fairness, and interference [15,27–30]. For example, efficiently reassigning unused airtime resource to those in need (work-conservation) is critical. When two clients are known to have hidden interference, we can avoid scheduling the two in the same window. We rely on the previous work and future work for the detailed design of the scheduling algorithm and this

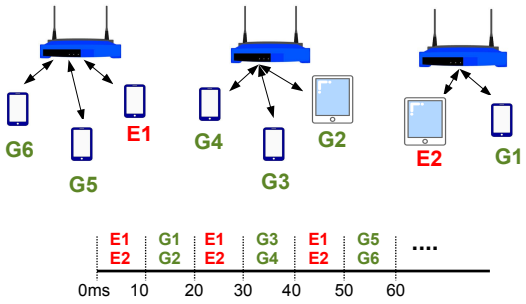


Figure 3: pTDMA scheduling example.

paper discusses high-level principles and constraints specific to running pTDMA in WLANs. Ultimately, the scheduling mechanism is up to each network operator who can program the algorithm in the global controller.

Because the pTDMA qdisc operates above the WiFi driver, we can't tightly control per-packet transmission timing to a microsecond level as conventional TDMA does. Also, the tight TDMA scheduling can perform worse in the presence of unexpected interference or burst traffic, which 802.11 CSMA MAC is supposed to deal with. Thus, *pTDMA scheduling unit is not per-packet basis but is a larger time window during which a client can transmit and receive multiple packets.* (As 802.11 MAC aggregation typically use 4ms time limit, this could be a good minimum limit. We use 10ms in our prototype.) Because perfectly estimating future traffic pattern and demand is difficult, we also schedule multiple clients in a common window to help maximize channel utilization while controlling the number of contending clients. Finding a good balance between multiplexing gain and contention overhead has been studied in ref. [31] and the pTDMA scheduler can leverage the previous study.

pTDMA should carefully determine the interval between two consecutive time windows of a client to meet currently active application's delay and jitter requirement. Consider an example when video streaming and VoIP applications, running on the same client, have similar inter-burst intervals but their bursts are interleaved. pTDMA can delay one flow to match its burst pattern to the others' and schedule the client's ontime windows to fit the matched pattern; delaying the streaming flow is more desirable because a streaming buffer can absorb additional delay while VoIP has a more stringent delay requirement. Keeping the interval time constant is desirable to control jitter and to provide consistent TCP performance. In addition to the interval, the window size (scheduling unit) must be set carefully to meet various application requirements, while balancing channel utilization and contention overhead.

Optimally deciding the window size & interval and scheduling clients over windows is a challenging problem, especially when there are many network slices and client devices. In this case, scheduling all clients of each network instance in a window (the baseline approach) can simplify the problem and increase the chance to meet the basic requirement – separate airtime slice for each network instance – while minimizing the interval time.

4.3 Downlink Control and Power-Saving

For AP-to-client downlink control, we may implement a similar pTDMA scheduler on APs but its implementation may be complicated because the AP has to control timings for every client. As an alternative, we found WMM-Power Save (WMM-PS) mechanism that triggers AP's downlink transmission of buffered data to a client by the client's uplink transmission [32]. (This is different from legacy power saving, in which a client typically waits till the next beacon to transmit or receive.) WMM-PS is a part of Wi-Fi Certification program and implemented in most client devices. *We leverage WMM-PS to indirectly confine downlink transmissions to the time window controlled by the client's pTDMA scheduler.*

Typically WMM-PS is used for VoIP or video traffic that has a regular burst pattern. Best effort applications (email, web and file transfer) are handled by legacy power saving. When backlogged packets are present for TX or RX, e.g., from bulk file transfer, most WiFi drivers stay in the Constant Awake Mode (CAM), as opposed to power saving mode, even when they do not obtain constant channel access due to contentions. Because pTDMA controls contention in each window, it allows the WiFi interface to more efficiently utilize its active time and to sleep longer outside of the ontime window. Thus, pTDMA makes WMM-PS also attractive for best effort traffic. In addition, *meSDN* OVS can detect two flows with their burst patterns interleaved and combine their patterns appropriately in the pTDMA qdisc such that the inter-burst time of the combined flow is maximized, increasing the sleeping time. In addition, we will show *pTDMA and WMM-PS can improve power-efficiency without losing throughput performance.*

5. PROTOTYPE IMPLEMENTATION

In this section, we describe a scaled-down version of *meSDN* framework to prototype the pTDMA service on top.

5.1 Architecture

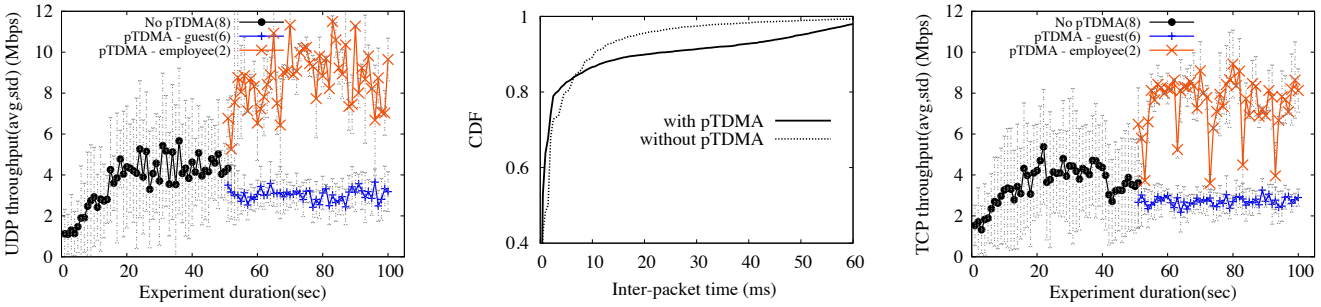
We implemented a pTDMA scheduler using Linux multiq [33] qdisc as a basis. pTDMA qdisc represents the *meSDN* extension of the *meSDN* architecture (Fig. 2). Implementing it in the driver could give tighter control over airtime usage, but due to practical issues of less accessibility we prototyped in the qdisc.

In Flow Manager, we use OVS to measure the per flow statistics and also take QoS actions on per-flow per-app. We also prototyped the OpenFlow-Wireless extension by conveying WiFi stats (such as RSSI) in Linux packet buffer (`skbuff`) forwarded from the driver to the OpenFlow datapath but didn't use the stats for pTDMA scheduling in this implementation. Finally, the Global Controller communicates with the local controller to set the schedule for pTDMA qdisc.

5.2 Challenges

Millisecond level synchronization, instead of microsecond required by conventional per-packet TDMAs, is needed to enforce pTDMA schedules across clients. Mobile phone GPS or NTP can provide such accuracy as we observed from our Android phones.

Driver buffering delay is known to be large enough to cause application performance to drop under certain conditions (Bufferbloat [34]) because WiFi (and also Ethernet) drivers typically have a fairly large (100-300 or more) ring



(a) UDP throughput (b) Guest1 UDP packet interval (c) TCP throughput

Figure 4: Testing with 2 employees and 6 guest devices with 50:50 airtime share.

packet buffer. The time to drain the buffer can take hundreds of milliseconds or more when the buffer is filled with max MTU sized packets and the wireless throughput is slow. This can hurt pTDMA because the driver may consume more airtime than scheduled to drain all the buffered packets even after pTDMA qdisc stops. As a solution to bufferbloat, Ethernet drivers started to implement Byte Queue Limits (BQL), where the limit is dynamically set based on number of bytes the NIC dequeued recently [35]. In our prototype, we took a similar BQL approach into the WiFi driver while providing reasonable buffer space for 802.11 MAC aggregation.

802.11 beacons: we believe pTDMA scheduling cycle doesn't have to be tied to the beacon interval, which can differ across APs. We don't control or coordinate beacon timing, but rather inform the beacon TX schedule to the pTDMA scheduler, which then leaves some time after each beacon, for multicast traffic and unicast traffic for legacy power saving devices.

5.3 Evaluation

We prototyped *meSDN* client-side components on eight Google Nexus 4 Android phones. We formed two network slices – an "employee" network with 2 devices and a "guest" network with 6 devices – and applied the same pTDMA schedule of Fig. 3 providing 50:50 airtime share to the two virtual slices, but with all 8 devices connected to one physical AP.

In Fig. 4(a), all devices are sending uplink iperf UDP at 12 Mbps, and the pTDMA scheduling is initiated at 50 sec. The graph plots the average throughput over the devices in each network slice with the high and low water-mark bars presenting the standard deviations. The first 20 seconds are impacted by low wireless speeds from fresh starts and were excluded in the following analysis. We clearly see the employee average is almost 3X larger than the guest average, indicating pTDMA provides the intended airtime share. pTDMA also improves fairness with lower temporal variations compared to non-pTDMA, as pTDMA reduces the number of simultaneously contending nodes in a window (from 8 to 2 in the employee network slice and from 8 to 6 in the guest slice). As we have only two employee devices, the "employee" slice shows higher statistical variation. Guests (6 devices) have much smaller variation than non-pTDMA (8 devices) thanks to the reduced contention of pTDMA. The reduced contention also increased net throughput, by 3 to 10% for different schedules.

To assess power-saving improvement, we plot inter-packet transmission intervals of one guest device in Fig. 4(b). pTDMA gives longer interval time during which the WiFi interface can sleep. While at this time, we did not directly measure device power consumption, we compute the total time the interface would sleep assuming a 5 ms timer for WMM-PS to detect inactivity and go to the sleep state: it can sleep 80% of entire run time with pTDMA while only 28% without pTDMA. Fig. 4(c) shows that the increased transmission intervals in the pTDMA schedule do not adversely impact TCP performance.

We tested different schedules, e.g., only one employee in a window or three guests in a window, and observed very similar results. Some schedules increased aggregate throughput (by 10%) while deviating from the intended employee:guest throughput ratio, suggesting a need for future study.

6. RELATED WORK & DISCUSSION

WLAN infra virtualization: Current APs can host multiple SSID networks or 'virtual APs' (e.g., employee vs. guest) on an AP radio [36] as a basis to control authentication, security and Ethernet-side bandwidth but they don't guarantee wireless resource share to each SSID network. Throttling Ethernet bandwidth may indirectly control wireless airtime usage of uplink TCP traffic, but with limited granularity; uplink UDP can't be controlled at all [37]. Recent work of OpenRadio [7], OpenRAN [8], CloudMAC [38] and Odin [39] extends SDN control to wireless APs. The Wireless & Mobile Working Group in Open Networking Foundation (ONF) is investigating various use cases for SDN-enabled APs and planning to standardize SDN control of wireless APs by extending OpenFlow [40]. SDN-enabled APs can control wireless resource for downlink traffic. However none of these work are considering SDN on end devices or addressing uplink control issue, and therefore they cannot control uplink traffic. Uncontrolled uplink transmissions can interfere with other uplink traffic and downlink traffic, so SDN-enabled APs can not guarantee uplink airtime or even downlink airtime. By carefully tuning 802.11e QoS parameters on the AP, we can probabilistically control downlink airtime shares and be more immune to uplink traffic, but this breaks 802.11 QoS mechanism or limits the number of virtual networks to four (802.11 QoS classes) [41].

Client-side solutions: Implementing per-packet TDMA MAC in WiFi driver has been demonstrated to virtualize airtime [14] but its throughput and scalability is challenged by the lack of sub-millisecond level 1) hardware control from

the driver software and 2) clock synchronization across devices. SplitAP [15] loosely controls uplink airtime by shaping client's outbound traffic using Click router but it causes under-utilization of airtime. In contrast, pTDMA achieves tighter airtime control and also improves aggregate throughput and power-efficiency. The work in [11] deployed OVS on Android to efficiently utilize multiple network interfaces on a multi-homed device. They also introduced a local controller to manage OVS but the coordination with other devises or the rest of the SDN framework was not discussed.

Application-awareness: MultiNets [42] and Delphi [23] have proposed mechanisms to use multiple network interface of the smartphones based on the policies (i.e. energy saving, throughput performance, data usage cost, delay sensitivity), manually given by the user [42] or specified by each application [23]. Rather than relaying on users or apps to specify their objectives, *meSDN* monitors and analyzes network flows to learn the app's network demands real-time. pTDMA changes the transmission patterns of applications and thus may affect Quality of Experience (QoE) of application users. Recent work [43] shows the possibility of monitoring network flows to estimate the QoE, which *meSDN* can leverage to improve pTDMA scheduling.

Interference is another hurdle in achieving RF resource guarantees in unlicensed bands. pTDMA can control interference within and also between *meSDN* network slices. External interferences can be handled by existing work on interference monitoring and mitigation [19–21]. Enterprise Radio Resource Management solutions today aim to realize such centralized schemes by tightly monitoring and reacting to interference at down to 4 min adaptation cycle [17]; recent small business and home APs are also controlled by 'cloud' for better radio resource management [18]. Even when external interference is unmanaged, pTDMA can still control the airtime 'share' among network slices, while each share will experience the external interference and deal with it based on 802.11 MAC.

Client WiFi driver modification: Though there have been many innovative solutions that require "only" driver changes [14, 30, 44], none of them have been widely adopted. We found that end-device manufacturers tend to avoid making any custom changes on the drivers shipped by the chipset vendors because it is hard to maintain the custom changes throughout the future chipset/driver releases and it restricts flexibility of end-device manufacturers in selecting radio chipsets for future device designs. We believe that the WiFi protocol and the client WiFi stack are better to be kept intact to develop and deliver the new *meSDN* framework reliably across a multitude of end devices. Hence, our current prototype leveraged Linux standard components – Qdisc and OVS – as building blocks. Both components exist in recent Android kernel source.

WiFi community may recognize the need for tighter airtime control and introduce additional control knobs in the driver (e.g. millisecond-level control of sleep/awake) or implement existing optional QoS features from the 802.11 standard. They will surely benefit *meSDN*, and we look forward to seeing such new knobs. However, even with those new knobs in the driver, we believe pTDMA scheduling intelligence is better to reside outside the WiFi driver/stack and rather in the *meSDN* global or local controller where information about user applications and other devises is available.

Incremental deployment: We had to root the Android devices to install OVS and pTDMAqdisc kernel modules. (On the other hand, changing the WiFi driver required reimaging the entire kernel.) To ensure that heterogeneous end devices can participate in and benefit from *meSDN*, the kernel modules need to be natively integrated into the stack OS by the device manufacturers and similar support from other operating systems are needed.

To support hybrid deployments and phased migrations, *meSDN* and non-*meSDN* end devices will need to co-exist in the same WLAN environment especially in non-enterprise settings where synchronized deployment over all client devices is difficult. To continue to provide air time guarantees to the *meSDN* clients, non-*meSDN* clients can be controlled via mechanisms proposed earlier, such as dropping 802.11 data frames to lead them to backoff.

7. CONCLUSION

In this paper, we argued that extending SDN capability to mobile end devices can provide true end-to-end software defined solutions for many network problems such as QoS, virtualization, and fault diagnosis. We proposed the meSDN (mobile extension of SDN) framework and demonstrate its use-cases: application-aware 802.11e QoS, OpenFlow-based fault diagnosis and WLAN virtualization. As a proof-of-concept, we used meSDN framework to implement a WLAN virtualization service that effectively guarantees airtime shares to network slices using a Time Division Multiple Access like airtime scheduling, named pseudo TDMA (pTDMA).

8. REFERENCES

[1] E. Cuervo *et al.*, "MAUI: Making Smartphones Last Longer with Code Offload," ACM MobiSys 2011.

[2] B.-G. Chun *et al.*, "CloneCloud: Elastic Execution Between Mobile Device and Cloud," ACM EuroSys 2011.

[3] UCI Forum, "UC SDN Use Case – Automating QoS." `http://tinyurl.com/lch8qy7`.

[4] W. Kim *et al.*, "Automated and scalable QoS control for network convergence," USENIX INM/WREN, 2010.

[5] "Open vSwitch." `http://openvswitch.org/`.

[6] A. Patro, S. Govindan, and S. Banerjee, "Outsourcing Home AP Management to the Cloud through an Open API.," Open Networking Summit, 2013.

[7] "OpenRadio." `http://gigaom.com/2012/04/19/openradio-changes-what-it-means-to-be-an-isp/`.

[8] M. Yang *et al.*, "OpenRAN: A Software-deïñĄned RAN Architecture Via Virtualization," ACM SigComm Poster, 2013.

[9] J. Dennis Gessner *et al.*, "Towards a User-Friendly Security-Enhancing BYOD Solution.," tech. rep., NEC Europe Ltd., 2013.

[10] "Riverbed Steelhead Mobile." `http://tinyurl.com/lka49nb`.

[11] K.-K. Yap *et al.*, "Making use of all the networks around us: a case study in android," ACM CellNet, 2012.

[12] M. Balon and B. Liau, "Mobile virtual network operator," in *NETWORKS, 2012*, 2012.

[13] Y. Yiakoumis *et al.*, "Slicing home networks," ACN HomeNets, 2011.

[14] G. Smith *et al.*, "Wireless virtualization on commodity 802.11 hardware," ACM WinTECH, 2007.
[15] G. D. Bhanage *et al.*, "SplitAP: Leveraging Wireless Network Virtualization for Flexible Sharing of WLANs," in *IEEE GLOBECOM*, 2010.
[16] "Radio Resource Management under Unified Wireless Networks.," tech. rep., Cisco, August 2007.
[17] "Configuring Adaptive Radio Management (ARM) Profiles and Settings," tech. rep., 2008.
[18] "Meraki White Paper: Meraki Hosted Architecture." `http://tinyurl.com/kb4dsyl`.
[19] V. Shrivastava *et al.*, "PIE in the sky: online passive interference estimation for enterprise WLANs," USENIX NSDI, 2011.
[20] S. Rayanchu, A. Patro, and S. Banerjee, "Catching whales and minnows using WiFiNet: deconstructing non-WiFi interference using WiFi hardware," USENIX NSDI, 2012.
[21] E. Rozner *et al.*, "Traffic-aware channel assignment in wireless LANs," IEEE ICNP, 2007.
[22] Y. Yiakoumis *et al.*, "Putting home users in charge of their network," ACM UbiComp, 2012.
[23] S. Deng, A. Sivaraman, and H. Balakrishnan, "All Your Network Are Belong to Us: A Transport Framework for Mobile Network Selection," in *ACM HotMobile*, 2014.
[24] Z. Qazi *et al.*, "Application-Awareness in SDN," ACM SigComm Demo, 2013.
[25] F. Qian *et al.*, "Periodic Transfers in Mobile Applications: Network-wide Origin, Impact, and Optimization," in *ACM WWW*, 2012.
[26] J. Huang *et al.*, "A Close Examination of Performance and Power Characteristics of 4G LTE Networks," in *ACM MobiSys*, 2012.
[27] I. Rhee *et al.*, "DRAND: distributed randomized TDMA scheduling for wireless ad-hoc networks," ACM MobiHoc, 2006.
[28] S. Borst, "User-level performance of channel-aware scheduling algorithms in wireless data networks," *IEEE/ACM Trans. Netw.*, vol. 13, no. 3.
[29] H. Kim and Y. Han, "A Proportional Fair Scheduling for Multicarrier Transmission Systems," *IEEE Communications Letters*, vol. 9, no. 3, 2005.
[30] V. Shrivastava *et al.*, "CENTAUR: realizing the full potential of centralized wlans through a hybrid data path," ACM MobiCom, 2009.
[31] I. Rhee, A. Warrier, M. Aia, and J. Min, "Z-MAC: a hybrid MAC for wireless sensor networks," ACM SenSys, 2005.
[32] "WMM^{TM} Power Save for Mobile and Portable Wi-Fi®CERTIFIED Devices.," tech. rep., Wi-Fi Alliance, 2005.
[33] "HOWTO for multiqueue network device support." `http://www.mjmwired.net/kernel/Documentation/networking/multiqueue.txt`.
[34] J. Gettys and K. Nichols, "Bufferbloat: Dark Buffers in the Internet," *ACM Queue*, vol. 9, no. 11, 2011.
[35] "Network transmit queue limits." `https://lwn.net/Articles/454390/`.
[36] K.-K. Yap *et al.*, "Separating authentication, access and accounting: A case study with OpenWiFi," tech. rep., OpenFlow TR 2011-1.
[37] K. Cai *et al.*, "A Wired Router Can Eliminate 802.11 Unfairness, But It's Hard," ACM HotMobile, 2008.
[38] J. Vestin *et al.*, "CloudMAC: torwards software defined WLANs," ACM MobiCom, 2012.
[39] L. Suresh *et al.*, "Towards programmable enterprise WLANS with Odin," ACM HotSDN, 2012.
[40] Open Networking Foundation (ONF), "Wireless & Mobile Working Group." `http://tinyurl.com/ltud7s2`.
[41] N. Kiyohide, S. Yozo, and N. Nozom, "Airtime-based Resource Controls in Wireless LANs for Wireless Network Virtualization," IEEE ICUFN, 2012.
[42] S. Nirjon *et al.*, "MultiNets: Policy Oriented Real-Time Switching of Wireless Interfaces on Mobile Devices.," IEEE RTAS, 2012.
[43] V. Aggarwal *et al.*, "Prometheus: Toward Quality-of-Experience Estimation for Mobile Apps from Passive Network Measurements," ACM HotMobile, 2014.
[44] "Cisco Compatible Extensions Program for Wi-Fi Tags Guidelines.," tech. rep., Cisco, 2007.