PULS: Processor-Supported Ultra-Low Latency Scheduling

Simon Yau, Ping-Chun Hsieh, Rajarshi Bhattacharyya, Kartic Bhargav K. R.,
Srinivas Shakkottai, I-Hong Hou, and P. R. Kumar
Texas A&M University, College Station
{symoyau, lleyfede, rajarshibh, kbhargav, sshakkot, ihou, prk}@tamu.edu

ABSTRACT

An increasing number of applications that will be supported by next generation wireless networks require packets to arrive before a certain deadline for the system to have the desired performance. While many time-sensitive scheduling protocols have been proposed, few have been experimentally evaluated to establish realistic performance. Furthermore, some of these protocols involve high complexity algorithms that need to be performed on a per-packet basis. Experimental evaluation of these protocols requires a flexible platform that is readily capable of implementing and experimenting with these protocols.

We present PULS, a processor-supported ultra low latency scheduling implementation for testing of downlink scheduling protocols with ultra-low latency requirements. Based on our decoupling architecture, programmability of delay sensitive scheduling protocols is done on a host machine, with low latency mechanisms being deployed on hardware. This enables flexible scheduling policies on software and high hardware function re-usability, while meeting the timing requirements of a MAC. We performed extensive tests on the platform to verify the latencies experienced for per packet scheduling, and present results that show packets can be scheduled and transmitted under 1 ms in PULS. Using PULS, we implemented four different scheduling policies and provide detailed performance comparisons under various traffic loads and real-time requirements. We show that in certain scenarios, the optimal policy can maintain a loss ratio of less than 1% for packets with deadlines, while other protocols experience loss ratios of up to 65%.

CCS CONCEPTS

• **Networks → Programmable networks**; *Wireless access networks*;

KEYWORDS

MAC Scheduling, Ultra-low latency, Software Defined Radio

ACM Reference Format:
Simon Yau, Ping-Chun Hsieh, Rajarshi Bhattacharyya, Kartic Bhargav K. R., Srinivas Shakkottai, I-Hong Hou, and P. R. Kumar. 2018. PULS: Processor-Supported Ultra-Low Latency Scheduling. In *Mobihoc '18: The Nineteenth International Symposium on Mobile Ad Hoc Networking and Computing, June 26–29, 2018, Los Angeles, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3209582.3209609

1 INTRODUCTION

Strict latency requirement for some flows, while maintaining high overall system throughput, is currently one of the most critical challenges for next-generation wireless networks. Emerging applications, such as virtual reality (VR) [1], factory Internet of Things (IoT), and tactile Internet [9], require an end-to-end latency between 1 to 10 milliseconds (ms) to provide seamless user experience. Other traffic might simply require high throughput, such as data downloads, or intermittent connectivity with low data rates, such as IoT applications. However, existing wireless networks cannot provide such stringent latency guarantees, especially with heterogeneous networks consisting of real-time and non-real-time traffic. For example, the round-trip time of LTE is estimated to be at least 20 ms, including the transmission time, scheduling overhead, and processing delay [5]. In practice, the current LTE technology can only support voice or video streaming applications with round-trip time in the range of 20-60 ms [14]. For Wi-Fi networks, due to the nature of random access, the round-trip time could vary from several ms to hundreds of ms depending on the traffic load [22]. Therefore, compared to the current technology, the latency budget is expected to be at least one order of magnitude smaller in next-generation wireless networks.

To provide strict per-packet latency guarantees, numerous theoretical solutions have been proposed to accommodate *per-packet deadline constraints* in wireless scheduling. For example, [7] proposes a theoretical framework to study wireless scheduling with per-packet deadline constraints. In this framework, packets not delivered on time are dropped, and the goal is to attain an average percentage of on-time deliveries. Later on, this framework has been applied to many other scenarios, such as utility maximization [8], scheduling for both latency-constrained and best-effort traffic [10], broadcast traffic [6], and multicast traffic [11]. The performances of these protocols are usually measured by *timely-throughput*, i.e. the time average of the amount of data delivered within their deadlines. While the above proposals are promising, there has been no implementation for these ultra-low-latency wireless protocols while supporting other types of traffic. As such, we do not know what kinds of system throughputs and capacity regions are achievable with these systems. Without knowledge of the throughputs and capacity regions, these protocols may not function as expected.

Furthermore, future networks are expected to support many heterogeneous applications. Such diversity necessitates a system that is capable of switching between a set of MAC protocols for different applications. For example, one might choose to use backpressure/max-weight scheduling for throughput maximization [4, 23], deadline

Mobihoc '18, June 26–29, 2018, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5770-8/18/06…$15.00
https://doi.org/10.1145/3209582.3209609

constrained scheduling for latency sensitive flows [7, 17] or perhaps a time division solution to support polling of IoT devices or in vehicular communication [12, 13]. Thus, a range of scheduling algorithms need to be instantiated, with the scheduling algorithm that is to be applied to a particular aggregate of packets being chosen based on its requirements. This can be difficult to implement on hardware due to the development time required, and the fact that instantiating each algorithm occupies hardware resources (on either FPGA or ASIC) that cannot easily be shared.

Another constraint appears in the form of the computational resources needed to support these algorithms. For example, the Max-Weight Independent Set problem that needs to be solved repeatedly for scheduling in multihop wireless networks is known to be NP-hard, and even polynomial approximations need significant computational resources [21]. More recently, scheduling algorithms that take a given schedule and "puncture" it with delay sensitive packets have been proposed to increase the efficiency of the MAC for 5G applications, and these algorithms also require significant computational resources [2]. As machine learning approaches make their way into MAC algorithms, the need for high speed computation at each scheduling decision will only become more pressing. Hence, the desire to support a diverse range of scheduling algorithms, coupled with the fact that some of these algorithms might require significant computational resources suggests the need for a platform that can support processor-assisted software scheduling. Thus, to prototype these ultra-low-latency wireless protocols and achieve realistic timely-throughput and system throughput, a powerful software-defined radio (SDR) platform with dedicated architectural design is required to compute complex algorithms quickly, provide enough latency budget for checking deadlines, as well as minimize the software and hardware processing and interfacing overhead.

Our main contribution is to bridge the gap between theory and implementation for heterogeneous wireless networks supporting flows with strict per-packet latency constraints (real-time flows), as well as flows that have no latency constraints (non-real-time flows), while maintaining high overall system throughput by proposing PULS, a processor-supported software-defined wireless platform that can support ultra-low-latency scheduling protocols. The PULS platform consists of a host machine that has significant computational power in the form of a general purpose multicore CPU, coupled with an SDR platform with FPGAs for low-level processing. PULS aims to leverage the higher clock speeds, and memory available at the host machine (which are at least an order of magnitude higher than what is available on SDRs) for performing complex scheduling algorithms, while leveraging the deterministic performance of the FPGA while performing simple repetitive tasks associated with PHY and low-level MAC layers. To achieve the required per-packet latency performance while performing scheduling on the host, PULS needs to address the following challenges:

(1) **Low interfacing latency between software host and hardware.** There are three major factors that affect the end-to-end latency: (i) queuing delay on software host, (ii) interfacing latency between software host and hardware, and (iii) hardware processing time. Queuing delay depends mainly on the scheduling policy, and hardware processing time can usually be made small due to the high clock rate supported by current technology. Therefore, with a proper choice of scheduling policy and hardware component, interfacing latency between software host and hardware needs to be minimized in order to achieve ultra-low latency. In Section 5, we present a simple experiment that demonstrates the interfacing latency of PULS is indeed small compared to packet deadlines.
(2) **Enforce per-packet deadline on a software-defined wireless platform.** PULS aims to support per-packet latency as low as 1 ms. When packets arrive at software host, they are first queued and start waiting for transmission according to some scheduling policy. The deadline of each packet in the queue needs to be tracked and checked before transmission. A packet that misses its deadline should be dropped from the queue. Moreover, due to the nature of SDRs, packet transmission is carried out on hardware while packet scheduling is often done on software host.
(3) **Achieve realistic per-flow timely-throughput and overall system throughput.** Ultra-low latency needs to come with realistic timely-throughput. Given the same physical data rate, the overhead of enforcing per-packet deadlines could be quantified by the difference in total MAC-layer throughput between the networks with and without packet deadlines. However, this should not come at the cost of reduced system throughput. In Section 6.1, through an experimental study on system capacity, we show that PULS achieves almost the same MAC-layer throughput as that with no deadlines.
(4) **Support functions working on heterogeneous time scales.** MAC layer functions operate on very different time scales. For example, an ACK response needs to be done within tens of microseconds. The transmission time of a typical data packet is between 0.5 to 1 ms. The target per-packet deadline is between 1 to 10 ms. The parameters of wireless protocols usually change over a period of at least several seconds to several minutes. In Section 3, we describe the separation principles of PULS which inherently incorporates the heterogeneity in time scale.
(5) **Support various ultra-low-latency downlink applications.** For example, VR requires latency as low as 1 ms with moderate timely-throughput while factory automation needs ultra-low packet loss rate with latency of about 5-10 ms. PULS is able to support applications with totally different performance requirements and provide a programmable environment for different wireless protocols.

To tackle the above challenges, PULS follows three major design principles. First, as a software-defined wireless testbed, PULS addresses the heterogeneous time scales of MAC functions by applying a Host-FPGA separation principle by performing scheduling on the host, and low-level MAC processing on FPGA. Next, PULS uses a Mechanism-Policy separation for both flexibility and performance by ensuring that mechanisms (specific function blocks), are decoupled from the policy, the specification of how particular scheduling protocol should be performed. Third, to support a broad class of scheduling policies, we borrow ideas from both WiFi as well as LTE standards, and build up a set of basic MAC functions required by most of the wireless protocols.

The rest of the paper is organized as follows. Section 2 summarizes the related works on low-latency wireless networks. Section 3 describes the design principles of PULS. The implementation of PULS is detailed in Section 4. Section 5 discusses the interfacing latency. Section 6 provides an extensive experimental study on ultra-low-latency protocols. In Section 7, we provide possible research directions for future work. Finally, Section 8 concludes the paper.

2 RELATED WORK

Various SDR architectures have been proposed to mitigate the interfacing overhead between software host and hardware. Just to name a few, [18] proposes a split-functionality framework to significantly reduce the communication overhead between software host and hardware. Similarly, [3] introduces Decomposable Medium Access Control (MAC) Framework to identify basic functional components according to both timeliness and degree of code reuse. However, neither of them considers per-packet deadline constraints nor provides any experimental results for ultra-low-latency wireless networks.

Most of the existing experimental studies for wireless LANs focus primarily on maximizing system throughput or throughput-based network utility. For example, to achieve maximum throughput, the well-known backpressure algorithm has been tailored and implemented for various scenarios, such as multi-hop wireless networks [25], TDMA-based MAC protocol [15] and wireless networks with intermittent connectivity [20]. Besides, for wireless LAN with random access, [16] implements an enhanced version of 802.11 DCF and demonstrates that it achieves near-optimal throughput as well as fairness with the original DCF. However, all of the above studies provide no support for packets with latency constraints. To address latency requirement for industrial control applications, RT-WiFi, a WiFi-compatible TDMA-based protocol, has been proposed and implemented on commercial 802.11 interface cards [26]. However, it cannot achieve both ultra-low latency and satisfactory timely-throughput performance for each user at the same time due to the nature of TDMA.

On the cellular side, several preliminary studies about 5G provide candidate solutions to enhance the low-latency capability via either numerical and experimental evaluation. [28] studies the trade-off between latency budget and required bandwidth by applying the conventional OFDM framework to 5G networks through numerical analysis. However, these numerical results do not take the possible signaling and processing overhead into account. [29] provides experimental study for latency performance of 5G millimeter-wave networks with beam-forming. However, this solution relies heavily on the beam-tracking technique and frame structure employed by cellular networks and cannot be directly applied to wireless LAN applications. Besides, [19] demonstrates a wireless testbed that is potentially capable of supporting millisecond-level end-to-end latency requirement. However, it supports only single link and does not take wireless scheduling issue into account.

3 DESIGN OF PULS

Our objective was to implement ultra-low latency protocols that require scheduling on a per packet basis, with a focus on downlink protocols, while concurrently supporting flows that do not have deadlines. These protocols may include algorithms with a high complexity that may not be suitable for implementation on the FPGA. In this section, we will explain the basic design principles upon which PULS was built to achieve the goals described in the previous sections.

3.1 Basic MAC Functions for Wireless Scheduling

Our platform borrows ideas from both the WiFi and LTE standards. From the WiFi side, we used features such as Carrier Sense for loose synchronization between the nodes and some robustness to interference. We also use the same interframe space timing intervals as WiFi for transmission (and reception) of packets. Furthermore, ACKs are sent immediately following transmission of data packets after a SIFS period, which allows us to know in a short period of time whether the packet was transmitted successfully. On the other hand, we are using a completely centralized framework for scheduling the different queues, which is similar to what LTE does. By using ideas from WiFi and LTE, we are able to obtain a deployment that is both lightweight and can be incorporated more easily into our framework for ultra low latency scheduling.

3.2 Mechanism-Policy Separation

In our design of PULS, we utilize a mechanism-policy separation used in [27]. Mechanisms can be functions or blocks used to handle the operations needed for per packet transmissions over the network, whereas policy refers to the high-level specification of the scheduling protocol itself. This mechanism-policy separation builds on the framework of Wireless MAC Processors, introduced by Tinnirello et al in [24].

Each mechanism has a set of inputs, outputs, events, conditions to check and possible actions that can be performed. Inputs and outputs of a mechanism take the form of register values (e.g. channel state and average energy), or an array of bytes (e.g. my address and packet).

Mechanisms provide the set of actions, events and also act as condition checkers, whereas the policy specifies the set of enabling functions, the parameters for the conditions, the set of update functions and the transition relations for the state machine of the scheduling protocol. Having the distinction between the different mechanisms and its associated events, actions, and conditions allows us to design new mechanisms more cleanly and reuse previously developed mechanisms. In addition to the mechanisms used in [27] and in [24], we implemented mechanisms to allow for backing off, deadline checking, packet dropping, controlling the packet arrival process, as well as features to increment and decrement some notion of deficit according to user-specified conditions. Deficits can be used in various ways to achieve some performance guarantees in delay-sensitive traffic, or to control the ratio of service times between inelastic and elastic flows. We will focus more on these mechanisms since they allow us to achieve certain guarantees on delay-sensitive traffic. These mechanisms can also be changed at runtime, allowing us to switch between different protocols on-the-fly. The implementation details of these mechanisms are laid out in section 4. Note, in the design of our testbed, mechanisms are implemented both on the FPGA as well as on the host machine. Furthermore, the inputs and outputs for each mechanism can be modified accordingly to allow for cross-layer designs. For example, the update deficit mechanism can use the packet's MCS as an input for the function to increment and decrement the deficits.

3.3 Flexible MAC through Host-FPGA Separation

For flexible MAC scheduling decisions, we employed a Host-FPGA separation, where high-level MAC functions such as packet scheduling, and packet dropping, are done on the host, and low-level functions such as packet encoding/decoding, carrier sensing, ACK processing, and CRC checking are done on the FPGA. This is done to allow for easy changes in packet scheduling decisions, while still being able to achieve our latency requirements for the platform.

4 IMPLEMENTATION OF PULS

We will discuss the implementation details of PULS in this section, namely, the flow of events, and the design of mechanisms for the nodes, illustrated in Figure 1. On the host, a continuous loop checks whether whether the current time mod the inter-arrival time is equal to zero using a tick count call supplied by the LabVIEW library, which returns a tick count with millisecond resolution. If it is, then the packet generator will generate a specified number of data packets with random bytes to emulate data traffic. Even when generating packets, this loop is capable of running approximately ten times per millisecond when no other applications are running. These data packets are then prepended with header bytes that are IEEE 802.11a PHY layer compliant. In addition to those headers, we also add deadlines to the packets that have been generated before putting them in a queue.

While packets are being generated, there is another loop running concurrently that is responsible for updating the states of the queue based on the number of packets that have been generated and feedback about the previous transmission. To obtain feedback about the previous transmission, the host will poll two registers per queue; an ACK counter and a timeout counter. Each of these counters will increment when an ACK or a timeout is received respectively. (A timeout occurs if an ACK is not received within 75 microseconds of the end of a transmission). This loop also schedules the next queue for transmission and push the packet to the FPGA for transmission.

Since our platform focuses on downlink scheduling, the mechanisms shown here are geared towards that, although most can be re-used for uplink scheduling as well. Starting with the National Instruments 802.11 Application Framework, we implemented additional mechanisms on the Host machine and on a USRP-2953R for flexible scheduling protocols. To support packets with strict deadlines, PULS presents a data transmission procedure which enables per-packet scheduling on the host while keeping the FPGA design simple.

4.1 Packet Generation

For packet generation, we implemented a packet generation mechanism that allows three main parameters that can be tweaked to replicate the different kinds of traffic that a user might want to experiment with: 1) interarrival times, 2) probability that packets get generated, 3) number of packets that are generated. Packets can be generated with interarrival times up to 1 ms in resolution, with a certain number of packets being generated at each time. In addition to that, packets can be generated deterministically or stochastically. At each interrival time, users can specify the probability that a certain number of packets get generated. Furthermore, the number of packets can be set to be a fixed number, or can take on a random number based on a distribution. For our purposes, in the random case, we use a simple uniform distribution for the number of packets that are generated. The maximum number of packets that can be generated are specified by the user. These arrival patterns and rates are by no means exhaustive, but it is sufficient for us to justify the performance of the platform and of particular protocols that we've implemented. (The details of the results are in Section 6.)

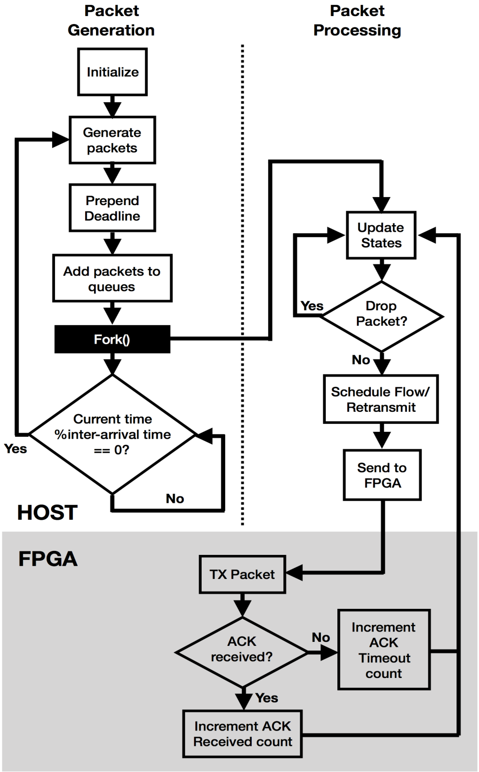

Figure 1: Architecture of PULS. Packet generation and Packet Processing happen at the host simultaneously (indicated by "Fork"), while low-level functionalities occur at the hardware (FPGA) level.

4.2 Queueing, Deadlines and Types of Flows

Each data flow, either elastic or inelastic, is associated with a queue on the Host. To add deadlines to the packets, we implemented a simple mechanism to prepend deadlines to each packet arriving at the queue. The input of this block is the current tick count of the system, the deadline, the incoming packet, and all the references to the queues associated with each flow. This block takes the current tick count of the system, adds the correct tick count corresponding to the deadline of the packet and prepends it to the packet before adding it to the correct queue. For inelastic flows, when a packet is generated (according to the Packet Generation mechanism), the corresponding *deadline identifier*, in the format of Host reference timer, is prepended onto the packet and then the packet is queued for scheduling. Since the absolute packet deadlines of each flow are assumed to form a non-deceasing sequence, every queue on the Host is always inherently arranged in an earliest-deadline-first manner. (Note, this may not be true if packet deadlines are changed frequently in a non-increasing manner. However, applications typically just require a baseline performance guarantee so this is valid for most cases. In the event that deadlines for packets are not arranged in an earliest-deadline-first manner, we can sort the packets while packets are being transmitted.) On the other hand, for elastic flows, the packet deadlines are set to be negative one for simplicity and better modularity in design.

4.3 Scheduling and Transmission Procedures

Once packets are generated and the packets are in their respective queues, scheduling is performed and repeated on a per-packet basis. In PULS, scheduling is aided by the use of several mechanisms to achieve our desired latency goals. First, we have packet dropping mechanism which scans the head-of-line packet of each queue, and drops the packets that have expired. This block only requires the references of each queue, and the current tick count of the system as inputs. It also has an output to inform other mechanisms whether or not a packet has been dropped. Next, we have a mechanism to update the state of the flows. Its job is to update the deficits associated with each flow based on whether or not an ACK was received and if a packet has been dropped from the queue. It uses the references of the queues, ACK received count, ACK timeout count, and the output of the packet dropping mechanism as its inputs. Lastly, we have a scheduling decision block that decides which flow to schedule based on the current states of the flows. This mechanism uses the deficits associated with each flow, the ACK counters for each of the flow, and the output of the packet dropping mechanism as inputs. All scheduling-related mechanism blocks are executed on the Host.

Scheduling and transmission executions are triggered when at least one queue is non-empty, and the scheduling state is UNLOCKED. The scheduling state becomes LOCKED as soon as a packet has been scheduled and is being processed for transmission. The state becomes UNLOCKED again when either of the two following FPGA events happen: an ACK reception or an ACK timeout. A successful ACK reception happens when an ACK packet is decoded successfully. This in turn increments the number of ACKs that have been received. The Host machine polls the register that stores the number of ACKs that have been received, and when the numbers differ from one iteration of the while loop to the next, the Host registers an ACK reception. For ACK timeouts, we implemented a mechanism that starts a counter after a packet has been transmitted on the FPGA. If an ACK is not received within a specified time period (set to be 75 microseconds), the count for timeout is increased on the FPGA. As in the case of ACK reception, if the count for ACK timeouts that the Host polls from the FPGA differ from one while loop iteration to the next, an ACK timeout event is registered. The timing of loop executions will be described further in Section 5. In every loop cycle, exactly one packet is scheduled for transmission. Before making a scheduling decision, the Host first "cleans up" the queues and updates the deadline-related state information by dropping expired packets in each queue using the mechanisms detailed above. Given the computing power of the Host, this cleanup can be finished almost instantaneously compared to the transmission time of a packet. (We should note that there will be more overhead if a batch of packets have expired, since we are only dropping one packet per loop execution, but this time is also negligible relative to the packet transmission time.) Given the queues in a clean state, the flow scheduler sets priorities of the flows according to the scheduling policy and schedules the flow with a non-empty queue and the highest priority. The scheduled queue then sends the head-of-line packet to the Prepare Interface Communication Protocol (ICP) Packet block, which removes the deadline identifier and appends the ICP header to the packet. The ICP header carries the information required by the FPGA, such as packet length, modulation and coding scheme, source MAC address, destination MAC address, and flow identifier, etc. Upon receiving the scheduled packet from the Host, FPGA simply triggers the required channel access procedure of the MAC layer as well as the physical transmission procedure in the PHY layer. By placing all the scheduling complexity on the Host, the design of PULS can be easily reproduced on an existing wireless interface card. The Access Point (AP) packet transmission procedure is summarized in Algorithm 1 and the function blocks associated with the transmission procedure are shown in Figure 2.

Table 1: Partial list of Mechanisms in PULS

Mechanism	Input	Ouptut	Description
Packet Generation	Interarrival times Probability of generation Max packets generated	Packets	Generates packets based on user specified parameters.
Prepend Deadlines	Current tick count Deadline Packets Queue references	N/A	Prepends deadlines to incoming packets and puts the packets into the correct queues.
Packet Dropping	Queue references Current tick count	Packet dropped?	Drops packets if deadlines have elapsed.
Update States	Queue references Packet dropped? ACK received count ACK timeout count Current state	Updated states	Updates the states associated with each flow.
Scheduling Decision	Current state Policy to use	Flow to schedule (or retransmit)	Decides which flow to schedule based on specified policy.
Retransmission	Current tick count Queue references ACK received count ACK timeout count Re-TX attempts	N/A	Stores transmitted packet, and retransmits if ACK was not received and deadline has not elapsed.

4.4 Retransmission

As reliable transmission is often required by mission-critical low-latency applications, PULS also supports retransmission for both elastic and inelastic flows to recover packet losses. Note that while MAC-layer retransmission is usually implemented in the hardware for conventional SDRs to minimize latency, the retransmission block of PULS is located in the Host for two reasons: (i) Packet deadlines need to be checked before any retransmission. Since deadlines are tracked in the Host, it is straightforward to handle retransmission in

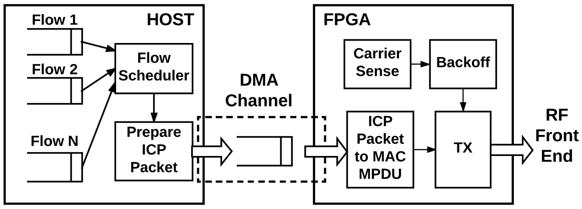

Figure 2: Packet transmission on PULS.

the Host. (ii) The Host-to-FPGA interfacing latency is low enough for supporting retransmission in the Host.

In PULS, retransmission is built on top of the scheduling and data transmission procedure described in Section 4.3. An additional retransmission mechanism with retransmission queue is created on the Host for temporarily storing the duplicate of the scheduled packet in the current loop cycle. The inputs to this block are the references to all the queues, the ACK counts for the flows, and the maximum retransmission attempts which can be easily configured in the Host according to the user specified scheduling policy. If an ACK timeout is received and the maximum retransmission count is not met, then the scheduler will attempt to retransmit the same packet in the next cycle; otherwise, the duplicate packet is removed from the retransmission queue.

The ICP packet is sent to the SDR's FPGA fabric via a Direct Memory Access (DMA) Channel. All PHY layer processing required to transmit the packet is performed on the FPGA using the Intellectual Properties (IP) provided by the 802.11 AF. While the platform supports mechanisms for random backoff, we set backoffs to zero since our scheduling algorithms are completely centralized.

After the packet is transmitted, the Received Packet mechanism will determine whether an ACK was received or not. If an ACK was received, then ACK count will be incremented. Otherwise, the ACK timeout count will be incremented. Received packets are then sent back to the Host for processing, again via a DMA Channel. Concurrently, the Host polls the FPGA registers for transmitted packet count, ACK count, and ACK timeout count, and updates the deficits accordingly based on these counts.

Algorithm 1: AP transmission procedure.

```
Initialize the state variables and the Host queues;
while station is ON do
    update Host queues and state variables;
    schedule the flow with the highest priority;
    send the scheduled packet to Prepare ICP Packet;
    state of flow scheduler ← LOCKED;
    while state of flow schedule is LOCKED do
        if receive ACK response then
            state of flow scheduler ← UNLOCKED;
        end
    end
end
```

5 MEASURING HOST-TO-FPGA INTERFACING LATENCY AND ROUND-TRIP LATENCY

The interfacing latency between hardware and software significantly affects the achievable link throughput of a software-defined wireless testbed. As discussed in Section 4, flow scheduling is repeated on a per-packet basis in a loop where scheduling-related function blocks are executed. The time between two consecutive loop executions depends on the round-trip transmission time of each packet plus the Host-to-FPGA interfacing latency. The interfacing latency between Host and FPGA needs to be low enough to support a packet deadline as low as 1 ms. Meanwhile, *round-trip latency*, which is defined as the elapsed time between a packet arrival at the Host and the ACK reception of the packet, indicates the minimum achievable per-packet deadline of a wireless platform. To measure interfacing latency and round-trip latency, we devise a simple experiment by using the FPGA counter for the timestamps of the packet events. The connection used was a PCIe bus via an ExpressCard slot on the laptop connected to the USRP-2953R through an ExpressCard-MXI Interface Kit for USRP RIO.

The experiment can be summarized as follows:

(1) Test packets of fixed payload size arrive at the Host periodically. The period is set to be large enough such that at each time there is only 1 test packet waiting for transmission in the Host queue. This completely eliminates the effect of queueing delay in the Host.
(2) When a test packet arrives at the Host, it is given a timestamp generated on the FPGA, denoted by t_h (read by the Host from an FPGA register) and then forwarded to the FPGA through a DMA Channel immediately.
(3) When FPGA detects the new test packet, FPGA starts processing the ICP header and retrieves t_h from the header. Along with the current FPGA counter denoted by t_f, the Host-to-FPGA interfacing latency can be derived as $t_f - t_h$.
(4) The packet is then transmitted. When the corresponding ACK is received, FPGA reads the current timestamp t_r and calculates the round-trip latency as $t_r - t_h$.

In the experiments, we measure both latency metrics for 50000 packets with a fixed interarrival time of 10 ms. Figure 3 shows the empirical cumulative distribution function (CDF) of the Host-to-FPGA interfacing latency for packets with 1500 bytes payload at a data rate of 54Mbps. The mean interfacing latency is around 192 μs, and the 90, 95, and 99 percentiles are 233.4, 257.6, and 316.1 μs, respectively. Table 2 further summarizes the statistics of the interfacing latency for different data rates and payload sizes. We see that both the mean and the percentiles of the Host-to-FPGA latency are almost invariant, regardless of data rate and payload size. Therefore, PULS indeed exhibits low and predictable interfacing latency.

Next, Figure 4 shows the empirical CDF of round-trip latency, and Table 2 summarizes the statistics of round-trip latency for different data rates and payload sizes. Since round-trip latency consists of both transmission time and Host-to-FPGA interfacing latency, it varies with the physical data rate and the payload size. For the six test cases listed in Table 2, the maximum 99 percentile round-trip latency is 933.7 μs. Therefore, PULS is indeed able to guarantee a

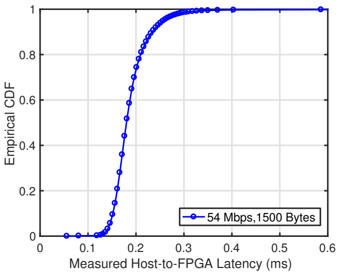

Figure 3: Empirical CDF of Host-to-FPGA latency for payload size = 1500 bytes and data rate = 54Mbps.

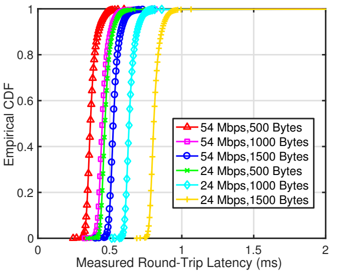

Figure 4: Empirical CDFs of round-trip latency for various data rates and payload sizes.

round-trip latency of less than 1 ms with high probability even for large packet sizes and moderate physical data rates.

Table 2: Host-to-FPGA latency results

Data rate (Mbps)	Payload size (bytes)	Host-to-FPGA latency (μs)			
		Mean	90%	95%	99%
54	500	185.2	227.6	251.5	301.8
54	1000	189.7	235	259	315
54	1500	192.2	233.4	257.6	316.1
24	500	187.8	228.4	252.7	305.7
24	1000	188.3	230.6	254.3	304
24	1500	189.2	231.5	255	304.6

Table 3: Round-trip latency results

Data rate (Mbps)	Payload size (bytes)	Round-trip latency (μs)			
		Mean	90%	95%	99%
54	500	377.0	419.2	443.4	494.4
54	1000	459.9	505.1	529.2	585.0
54	1500	536.9	578.5	602.3	660.8
24	500	479.5	520.2	544.4	598.1
24	1000	646.6	688.9	712.7	762.8
24	1500	817.9	860.2	883.9	933.7

6 EXPERIMENTAL RESULTS

We provide experimental results for a network with one AP and two downlink clients in various scenarios. Each client is associated with one real-time flow with per-packet deadlines as well as one non-real-time flow without deadline constraints. Each of the nodes (AP and clients) is a USRP-2953R that is connected to a Windows laptop acting as the Host machine. A specific scenario will have a certain arrival process, as well as predefined requirements for deadlines and delivery ratios for the real-time flows. These experiments were run using 1500B packets and IEEE 802.11a MCS 7 (54Mbps theoretical link data rate). While the packets are IEEE 802.11a PHY compliant, we do not consider all the features available in the standard. We consider four scheduling policies: Largest Deficit First (LDF) [7], Longest Queue First (LQF), Round Robin (RR), and Random. Based on the definition of deficit introduced in [7], LDF schedules the real-time flows with the largest deficit and selects non-real-time flows with the largest queue length if the real-time flows are empty. Ties are broken randomly. LQF, as the name suggests, schedules the flow with the longest queue, with ties broken randomly. Random policy randomly picks a flow to schedule among non-empty queues. RR schedules flows in the following order: real-time flow for client 1, real-time flow for client 2, non-real-time flow for client 1, and then non-real-time flow for client 2. If any of the queues are empty, it will schedule the next queue. A point to note is that dropping expired packets is not done in most implementations today. To ensure a fair comparison, we decided to enable packet dropping for all policies, which improves the performance for all policies.

The results are presented in this section as follows. In section 6.1, we show the capacity regions of a single client under the different policies for one scenario to show the achievable regions of our system. Then, in section 6.2, we present the throughputs of the policies under different scenarios to compare the system performance. In these sections, packets are generated every 5 ms and the number of packets generated are uniformly distributed from 0 to K_{RT} for real-time flows, and K_{NRT} for the non-real-time flows. We used a different arrival process in section 6.3 to have packets to arriving more frequently to show that ultra low latencies with 1 ms deadlines are indeed achievable.

6.1 Capacity Regions

For our capacity region experiments, we defined achievable regions as regions where the system deficit (accumulated when packets are dropped) and queue length of the clients does not grow to infinity. Per-packet deadline is set to 5 ms and the delivery ratio is set to 0.99, which means that the deficit increases by 0.99 every time a packet is dropped, and decreases by 0.01 when the packet is successfully delivered. In other words, if we require 99% of real-time packets to arrive in 5 ms, we say the policy can achieve that for any incoming packet rate where the deficits do not grow to infinity. As we can see from Figure 5, the LDF policy has a bigger achievable region than LQF, Random and RR. Random and RR has similar capacity regions, but Random performs slightly better than RR when K_{RT} is higher, and worse when K_{RT} is smaller. This is because the amount of time waiting for service is bounded for RR, so for small number of real-time packets, they will always be served before the packets expired. On the other hand, when K_{RT} is high, RR will not be able to serve the packets in time, but random has a chance of serving the

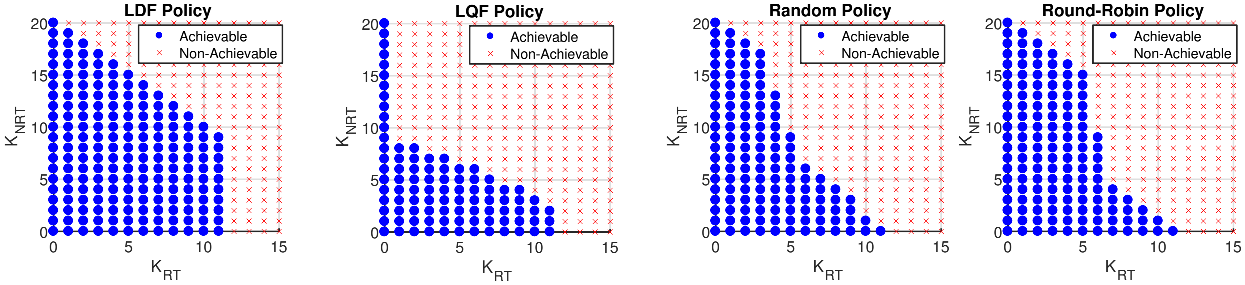

Figure 5: Capacity Regions for the different Policies with 5 ms deadline and 0.99 delivery ratio.

packets before they expire. LQF works better for higher value of K_{RT} than Random and RR, since it would schedule real-time flows more frequently, but suffers significantly when the value of K_{NRT} is higher than K_{RT}, since non-real-time flows will be schedule first. LDF always schedules real-time flows first, so the drop off in the capacity region is linear until K_{RT} = 11, after which the deficits start increasing to infinity for this delivery ratio and deadline. In this scenario, all policies can support up to a K_{RT} = 11, and serve up to 20 packets every 5 ms.

6.2 Throughput Performance

Using the capacity region plots, we can know what kind of arrival rates our platform is capable of supporting. However, this did not tell us much about system performance is affected, in terms of the throughputs for real-time and non-real-time flows when operating in different scenarios. For brevity, in this section, throughputs for real-time flows will be referred to as timely-throughput, and throughputs for non-real-time flows will just be referred to as throughputs. So, the next thing we did was to run experiments for multiple clients, each with a real-time and non-real-time flow, under various scenarios to see how each protocol performed. We ran two symmetric scenarios (clients have the same traffic and requirements), and two asymmetric scenarios. In the first scenario, we had K_{RT} = 4 and K_{NRT} = 4 for both clients, with deadlines set to 3 ms and delivery ratio set to 0.98. As we can see in Figure 6, while most protocols perform relatively well, LDF has a higher timely throughput and overall throughput for both clients. Next, we increased non-real-time traffic (K_{NRT} = 6) and decreased real-time traffic (K_{RT} = 3), and changed the requirements of the real-time flows to 2 ms deadlines and 0.95 delivery ratios to see how the system performs with lower latencies. This could be an example of two clients downloading a large file while streaming videos. As we can see in Figure 7, LQF has the worst performance since it will tend to serve non-real-time traffic first, followed by Random and RR. LDF has the best performance in both of these scenarios.

For the first asymmetrical scenario, we set K_{RT} = 3 and K_{NRT} = 5 for client 1, K_{RT} = 4 and K_{NRT} = 6 for client 2. For client 1, deadline of real-time packets was set to 2 ms, with 0.97 delivery ratio, whereas client 2 has a deadline of 3 ms with 0.98 delivery ratio i.e. client 1 has a real-time flow requirement where 97% of packets have to arrive in 2 ms and client 2 has a real-time flow requirement where 98% of packets arrive in 3 ms. As we can see in figure 8, LDF outperforms the other policies in terms of timely throughput and overall throughput. This difference is even more apparent when we 'increase' the asymmetry between the requirements of both clients. In our second asymmetrical scenario, we set K_{RT} = 7 and K_{NRT} = 4 for client 1, K_{RT} = 3 and K_{NRT} = 5 for client 2. Client 1's deadline was set to 5 ms with 0.98 delivery ratio, and client 2's deadline was 2 ms with 0.99 delivery ratio. This could be a case where client 1 is streaming a video where 5 ms latency is tolerable but it generates a lot of traffic, and client 2 was running a control application that does not produce as many packets but has stricter latency and delivery ratio requirements. We see that Random and RR does not have a good timely throughput for client 1, while LQF does not perform well for client 2. LDF outperforms all policies in both clients since it smartly schedules client 1 and client 2's real-time flows so that both requirements are met, without sacrificing overall system throughput. Note, since we are operating on the boundaries of LDF, the deficits of LQF, Random and RR are growing to infinity as well. The deficit evolution from the first symmetrical scenario is shown in Figure 11.

Table 4: Loss Ratios of Real-Time Flows

Arrival Rate	Deadline (ms)		Delivery Ratio		Policy	Loss Ratio (%)	
	C1	C2	C1	C2		C1	C2
K_{RT1} = 4	3	3	0.98	0.98	LDF	<2	<2
K_{NRT1} = 4					LQF	23.51	17.12
K_{RT2} = 4					Rand	15.17	15.62
K_{NRT2} = 4					RR	15.07	16.03
K_{RT1} = 3	2	2	0.95	0.95	LDF	<5	<5
K_{NRT1} = 6					LQF	60.24	54.97
K_{RT2} = 3					Rand	26.48	27.22
K_{NRT2} = 6					RR	19.96	21.27
K_{RT1} = 3	2	3	0.97	0.98	LDF	<3	<2
K_{NRT1} = 5					LQF	61.45	27.35
K_{RT2} = 4					Rand	27.2	18.07
K_{NRT2} = 6					RR	20.09	17.13
K_{RT1} = 7	5	2	0.98	0.99	LDF	<2	<1
K_{NRT1} = 4					LQF	7.18	63.51
K_{RT2} = 3					Rand	17.01	29.46
K_{NRT2} = 5					RR	19.14	21.42

Even though the throughput does not differ by much, the picture becomes very different when we examine the number of real-time packets that were dropped. Using the deficits of the clients, we calculated the loss ratios of the real-time flows for all the scenarios above. (Since the deficits of LDF stays at 0, the most we can conclude is the loss ratio is below 1 - delivery ratio). In Table 4, we see that the loss ratios of LQF, Random and RR are much higher than LDF. The biggest difference is for the fourth scenario, when LDF can maintain less than 1% loss rate for Client 2, but LDF, Rand and RR experience loss rates of 63.5%, 29.5% and 21.4% respectively.

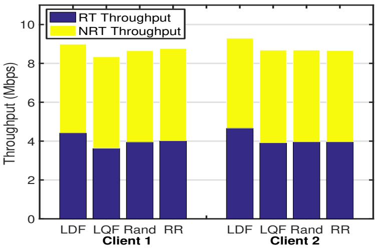

Figure 6: Throughputs for K_{RT}=4 and K_{NRT}=4.

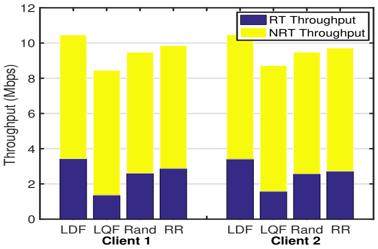

Figure 7: Throughputs for K_{RT}=3 and K_{NRT}=6.

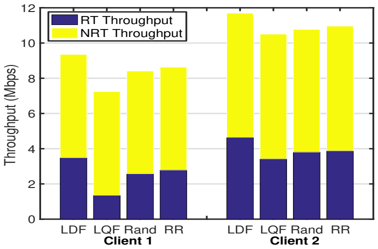

Figure 8: Throughputs for K_{RT1}=3, K_{NRT1}=5, K_{RT2}=4, K_{NRT2}=6.

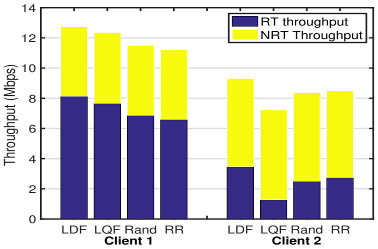

Figure 9: Throughputs for K_{RT1}=7, K_{NRT1}=4, K_{RT2}=3, K_{NRT2}=5.

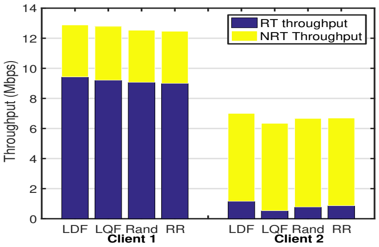

Figure 10: Throughputs for the arrival process with P(K_{RT1}) = 0.8, P(K_{NRT1}) = 0.3, P(K_{RT2}) = 0.1, P(K_{NRT2}) = 0.5.

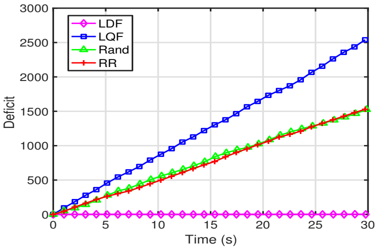

Figure 11: Deficit growth for the K_{RT}=4 and K_{NRT}=4 for both clients.

6.3 Changing the Arrival Process

In our final experiment, we modified the arrival process to further evaluate the policies under more stringent deadline requirements. Instead of packets being generated periodically every 5 ms, packets can now be generated every 1 ms. However, instead of generating it uniformly from 0 to K_{RT} (or K_{NRT}), 1 packet is generated with some probability, which is specified for each flow by the client. Thus, we move away from the frame-by-frame packet arrival process analyzed in [7] to a more general arrival process. While the proof of stability of the LDF algorithm under such arrivals in unknown, our system offers the ideal way of understanding its real-world performance.

Let P(K_{RTi}) and P(K_{NRTi}) be the probability of generating a packet for client i's real-time flow and non-real-time flow respectively. As in the previous experiment, the contrasts are most stark in the asymmetrical case, which is presented in the following scenario. Client 1 has P(K_{RT1}) = 0.8, P(K_{NRT2}) = 0.3 with the real-time flow requirement of 2 ms and 0.99 delivery ratio. This could be a user streaming a video while downloading some files in the background. Client 2 on the other hand has P(K_{RT2}) = 0.1, P(K_{NRT2}) = 0.5 with a deadline of 0 ms and 0.99 delivery ratio. Note that packets are only expired when the current time exceeds the deadlines, and hence even 0 ms deadlines give some throughput. This could be a mission-critical application which does not generate much traffic but requires control packets to be delivered within 1 ms. As we can see in Figure 10, client 1's timely throughput is lower for Random and RR and LQF causes client 2's timely throughput to suffer. However, LDF is able to support both clients and give good timely throughput and throughputs as well. Furthermore, this also shows that the system is capable of delivering packets under 1 ms.

These results show that our system is indeed capable of supporting experimentation of policies for ultra-low latency applications with various packet arrival patterns and deadline requirements.

7 FUTURE WORK

Crossing hardware-software boundaries always incurs interfacing overhead that can reduce the throughput of wireless systems. In our system, this overhead was in the order of hundreds of microseconds, which led to a throughput loss of about 30%. This is due to the fact that packets were being transferred with the scheduling decision through the use of DMA channels. DMA channels provide high throughput for data transfer at the cost of increased latency. To mitigate the effects of the interfacing delay overhead, one possible solution is to load packets onto separate queues on the FPGA via DMA channels, but convey the scheduling decision from the host to the FPGA through registers. Decisions can be written to an FPGA register in about 20 μs. This will reduce the interfacing delay overhead. However, this limits the decision space to the number of queues available on hardware, which could be significantly lower than the number of queues on software. In addition to that, it is not clear how queues on the host should be mapped to hardware queues. The other solution is to make several decisions at each decision stage e.g. decide which queues will transmit for the next three time slots. However, this will force protocols to commit their decision for the next few time slots which can lead to sub-optimal scheduling decisions. Neither of these solutions are fully compatible with existing policies, and we need new research to develop policies that work under these constraints as well.

Another method of obtaining both per-packet scheduling and mitigating the effects of interfacing latency is by using one-step look-ahead scheduling. Instead of scheduling and pushing a packet to the transmit queue only after receiving an ACK or timing out, we load two packets onto two separate queues during the current packet transmission; one for the case if the transmission was successful, and one if the transmission failed. The moment an ACK is received, or a timeout happens, the corresponding queue is scheduled on the hardware immediately. In this way, the effects of interfacing latency can still be mitigated while still allowing users the flexibility of per-packet scheduling. However, this also increases the complexity of the scheduler on the host.

8 CONCLUSION

Applications today have increasingly stringent requirements, especially in terms of latency and throghput. This presents next generation networks with one of its critical challenges: providing some measure of guarantee for applications with strict latency and throughput requirements. There exist theoretical frameworks to develop protocols that are able to do this, but there is still a gap between theory and implementation of these protocols. We aim to bridge this gap by developing PULS, which we have shown to be capable of supporting per-packet scheduling for downlink with latencies on the order of 1 ms, with realistic system throughputs. PULS was developed for with reprogrammability in mind, so new scheduling policies can be more easily implemented and experimented on it. Using PULS, we tested the performance of LDF, LQF, Random and RR under various scenarios and showed that LDF performs equally well or better than the other policies in all scenarios. The difference between LDF and the other policies are even more apparent when we observe the loss ratios of the policies under different traffic loads.

ACKNOWLEDGMENTS

This material is based upon work partially supported by NSF under Contract Nos. CCF-1619085, CNS-1646449, CNS-1719384, CNS-1149458, AST-1443891, Science & Technology Center Grant CCF-0939370, Office of Naval Research under Contract N00014-18-1-2048, and the U.S. Army Research Office under Contract No. W911NF-15-1-0279.

REFERENCES

[1] 2016. NSF Workshop on Ultra-Low Latency Wireless Networks. (November 2016).

[2] Arjun Anand, Gustavo de Veciana, and Sanjay Shakkottai. 2018. Joint Scheduling of URLLC and eMBB Traffic in 5G Wireless Networks. In *INFOCOM (to appear), 2018 Proceedings IEEE.*

[3] Junaid Ansari, Xi Zhang, Andreas Achtzehn, Marina Petrova, and Petri Mahonen. 2010. Decomposable MAC framework for highly flexible and adaptable MAC realizations. In *New Frontiers in Dynamic Spectrum, 2010 IEEE Symposium on.* IEEE, 1–2.

[4] Atilla Eryilmaz and R Srikant. 2006. Joint congestion control, routing, and MAC for stability and fairness in wireless networks. *IEEE Journal on Selected Areas in Communications* 24, 8 (2006), 1514–1524.

[5] Harri Holma and Antti Toskala. 2011. *LTE for UMTS: Evolution to LTE-advanced.* John Wiley & Sons.

[6] I-Hong Hou. 2015. Broadcasting delay-constrained traffic over unreliable wireless links with network coding. *IEEE/ACM Transactions on Networking* 23, 3 (2015), 728–740.

[7] I. H. Hou, V. Borkar, and P. R. Kumar. 2009. A Theory of QoS for Wireless. In *IEEE INFOCOM 2009.* 486–494.

[8] I-Hong Hou and PR Kumar. 2010. Utility maximization for delay constrained QoS in wireless. In *INFOCOM, 2010 Proceedings IEEE.* IEEE, 1–9.

[9] ITU-T. 2014. The Tactile Internet. (August 2014).

[10] Juan José Jaramillo and R Srikant. 2011. Optimal Scheduling for Fair Resource Allocation in Ad Hoc Networks With Elastic and Inelastic Traffic. *IEEE/ACM Transactions on Networking* 4, 19 (2011), 1125–1136.

[11] Kyu Seob Kim, Chih-ping Li, and Eytan Modiano. 2014. Scheduling multicast traffic with deadlines in wireless networks. In *INFOCOM, 2014 Proceedings IEEE.* IEEE, 2193–2201.

[12] Ray K Lam and PR Kumar. 2010. Dynamic channel partition and reservation for structured channel access in vehicular networks. In *Proceedings of the seventh ACM international workshop on VehiculAr InterNETworking.* ACM, 83–84.

[13] Ray K Lam and PR Kumar. 2010. Dynamic channel reservation to enhance channel access by exploiting structure of vehicular networks. In *Vehicular Technology Conference (VTC 2010-Spring), 2010 IEEE 71st.*

[14] Markus Laner, Philipp Svoboda, Peter Romirer-Maierhofer, Navid Nikaein, Fabio Ricciato, and Markus Rupp. 2012. A comparison between one-way delays in operating HSPA and LTE networks. In *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt), 2012 10th International Symposium on.* IEEE, 286–292.

[15] Rafael Laufer, Theodoros Salonidis, Henrik Lundgren, and Pascal Le Guyadec. 2011. XPRESS: A cross-layer backpressure architecture for wireless multi-hop networks. In *Proceedings of the 17th annual international conference on Mobile computing and networking.* ACM, 49–60.

[16] Jinsung Lee, Hojin Lee, Yung Yi, Song Chong, Edward W Knightly, and Mung Chiang. 2016. Making 802.11 DCF near-optimal: Design, implementation, and evaluation. *IEEE/ACM Transactions on Networking* 24, 3 (2016), 1745–1758.

[17] Ruogu Li, Atilla Eryilmaz, and Bin Li. 2013. Throughput-optimal wireless scheduling with regulated inter-service times. In *INFOCOM, 2013 Proceedings IEEE.* 2616–2624.

[18] George Nychis, Thibaud Hottelier, Zhuocheng Yang, Srinivasan Seshan, and Peter Steenkiste. 2009. Enabling MAC Protocol Implementations on Software-Defined Radios.. In *NSDI*, Vol. 9. 91–105.

[19] Jens Pilz, Matthias Mehlhose, Thomas Wirth, Dennis Wieruch, Bernd Holfeld, and Thomas Haustein. 2016. A Tactile Internet demonstration: 1ms ultra low delay for wireless communications towards 5G. In *Proc. of INFOCOM WKSHPS.* IEEE, 862–863.

[20] Jung Ryu, Vidur Bhargava, Nick Paine, and Sanjay Shakkottai. 2010. Backpressure routing and rate control for ICNs. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking.* ACM, 365–376.

[21] Sujay Sanghavi, Devavrat Shah, and Alan S Willsky. 2008. Message passing for max-weight independent set. In *Advances in Neural Information Processing Systems.* 1281–1288.

[22] Kaixin Sui, Mengyu Zhou, Dapeng Liu, Minghua Ma, Dan Pei, Youjian Zhao, Zimu Li, and Thomas Moscibroda. 2016. Characterizing and improving WiFi latency in large-scale operational networks. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services.* ACM, 347–360.

[23] Leandros Tassiulas and Anthony Ephremides. 1992. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. *IEEE transactions on automatic control* 37, 12 (1992), 1936–1948.

[24] Ilenia Tinnirello, Giuseppe Bianchi, Pierluigi Gallo, Domenico Garlisi, Francesco Giuliano, and Francesco Gringoli. 2012. Wireless MAC processors: Programming MAC protocols on commodity hardware. In *INFOCOM, 2012 Proceedings IEEE.* IEEE, 1269–1277.

[25] Ajit Warrier, Sankararaman Janakiraman, Sangtae Ha, and Injong Rhee. 2009. DiffQ: Practical differential backlog congestion control for wireless networks. In *INFOCOM 2009, IEEE.* IEEE, 262–270.

[26] Yi-Hung Wei, Quan Leng, Song Han, Aloysius K Mok, Wenlong Zhang, and Masayoshi Tomizuka. 2013. RT-WiFi: Real-time high-speed communication protocol for wireless cyber-physical control applications. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th.* IEEE, 140–149.

[27] Simon Yau, Liang Ge, Ping-Chun Hsieh, I Hou, Shuguang Cui, PR Kumar, Amal Ekbal, Nikhil Kundargi, et al. 2015. WiMAC: Rapid Implementation Platform for User Definable MAC Protocols Through Separation. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 109–110.

[28] Osman NC Yilmaz, Y-P Eric Wang, Niklas A Johansson, Nadia Brahmi, Shehzad A Ashraf, and Joachim Sachs. 2015. Analysis of ultra-reliable and low-latency 5G communication for a factory automation use case. In *Communication Workshop (ICCW), 2015 IEEE International Conference on.* IEEE, 1190–1195.

[29] Shohei Yoshioka, Yuki Inoue, Satoshi Suyama, Yoshihisa Kishiyama, Yukihiko Okumura, James Kepler, and Mark Cudak. 2016. Field experimental evaluation of beamtracking and latency performance for 5G mmWave radio access in outdoor mobile environment. In *Personal, Indoor, and Mobile Radio Communications (PIMRC), 2016 IEEE 27th Annual International Symposium on.* IEEE, 1–6.