SMUFF: Towards Line Rate Wi-Fi Direct Transport with Orchestrated On-device Buffer Management

Chengke Wang, *Peking University;* Hao Wang, *Shenzhen Kaihong Digital Industry Development Co., Ltd.;* Yuhan Zhou and Yunzhe Ni, *Peking University;* Feng Qian, *University of Southern California;* Chenren Xu, *Peking University, Zhongguancun Laboratory, and Key Laboratory of High Confidence Software Technologies, Ministry of Education (PKU)*

https://www.usenix.org/conference/nsdi24/presentation/wang-chengke

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation is sponsored by

SMUFF: Towards Line Rate Wi-Fi Direct Transport with Orchestrated On-device Buffer Management

Chengke Wang[ℙ], Hao Wang[𝕊], Yuhan Zhou[ℙ], Yunzhe Ni[ℙ], Feng Qian[ℂ], Chenren Xu[ℙℤ𝕂*]

[ℙ]*School of Computer Science, Peking University* [𝕊]*Shenzhen Kaihong Digital Industry Development Co., Ltd.*

[ℂ]*University of Southern California* [ℤ]*Zhongguancun Laboratory*

[𝕂]*Key Laboratory of High Confidence Software Technologies, Ministry of Education (PKU)*

Abstract – Wi-Fi direct transport provides versatile connectivity that enables convenient data sharing and improves the productivity of mobile end users. However, as today's smartphones are capable of near-Gbps wireless data rates, current solutions do not efficiently utilize the available bandwidth in this single-hop environment. We show that existing transport schemes suffer from resource-intensive reliable delivery mechanisms, inadequate congestion control, and inefficient flow control for achieving line-rate transmission in peer-to-peer Wi-Fi direct links. In this paper, we present SMUFF, a reliable file transfer service that achieves nearly the practical line rate of the underlying wireless bandwidth. We note a unique feature of direct transport – the sender can monitor each buffer along the data path and determine an optimal sending rate accordingly. Therefore, SMUFF can maximize throughput by strategically backlogging the appropriate amount of data in the bottleneck buffer. We have deployed SMUFF on four different phone models, and our evaluations with other transport schemes show that SMUFF achieves up to 94.7% of the practical line rate and 22.6% throughput improvement with a 37% reduction in CPU usage and a 15% reduction in power consumption, compared to state-of-the-art solutions.

1 Introduction

Peer-to-peer wireless connectivity is compelling for scenarios where quick and easy data sharing is critical. It not only improves data transport efficiency by reducing protocol overhead and backhaul traffic but also prevents potential data privacy leakage from the cloud. The industry has taken note of these advancements. An increasing number of smartphone manufacturers are introducing their own file transfer solutions, such as Apple AirDrop [1, 2], Huawei Share [3], Xiaomi Mobile Direct Fast Exchange [4], and Samsung Quick Share [5]. Looking ahead, these innovations will play a crucial role in supporting emerging mobile device applications. For instance, a VR headset may seamlessly load a 2 GB game asset from a PC [6], autonomous vehicles may exchange road traffic information via peer-to-peer wireless links [7], and devices involved in distributed ML training within robotic IoT networks or mobile federated learning may need to exchange model gradients over the wireless channel [8–10]. With such a wide range of applications, there is a need to unleash the full potential of peer-to-peer wireless data transfer.

With the ubiquitous deployment of Wi-Fi networks, Wi-Fi Direct [11] is an attractive local wireless data transmission technology to provide direct connectivity between devices. The standard claims that 802.11ac (Wi-Fi 5) already provides theoretical physical layer bandwidth up to 867 Mbps, and the practical wireless line rate can reach nearly 700 Mbps (§2.2). However, TCP (CUBIC), the most widely used transport scheme, averages only 442 Mbps on Wi-Fi Direct links, and other commercial solutions perform even worse (§2.3). Our further investigation reveals three key reasons that prevent the current transport schemes from achieving wireless line rate performance: *i)* The current reliable delivery mechanism uses a per-packet ACKing policy, which exacerbates channel contention on Wi-Fi links, and the compute-intensive network stack imposes more packet processing overhead. Both of these drawbacks significantly reduce throughput. *ii)* Today's widely deployed TCP (or QUIC) relies primarily on congestion control algorithms (CCAs) to probe and estimate the network bandwidth and adjust the sending rate, which is unsuitable for the one-hop single-flow scenario as it exhibits unnecessary startup phase and overreaction to the lossy wireless link. *iii)* The flow control mechanisms in existing transport schemes are inefficient in achieving line rate transmission in Wi-Fi Direct links as they are unaware of some on-path buffers and suffer from delayed ACK feedback and fixed parameter configurations. All of the above factors contribute to the reduced link bandwidth utilization.

In the presence of the above problems, we notice a distinctive characteristic of peer-to-peer direct data transmission that can be exploited to achieve high link utilization. Specifically, we are able to monitor the state of each individual buffer in the whole packet lifecycle because the two communicating devices are the only entities involved. This is in sharp contrast to a data center or Internet connection that relies on intermediate nodes (routers and switches) to forward the traffic – in this case, the sender typically needs to use a sophisticated CCA to *indirectly* infer the buffer states that are not available to the end host [12, 13] or obtain buffer states with the help of in-network devices [14–16]. In our scenario, the *direct* access to buffer states offers a unique opportunity to achieve a high data rate by backlogging data in the buffers.

*✉: chenren@pku.edu.cn

However, designing an efficient mechanism allowing a receiver to report the buffer state to the sender is not straightforward. First, existing flow control shipped with today's TCP and QUIC uses a per-packet ACK to allow the receiver to report its available space to the sender. However, a dense ACK stream can significantly reduce the throughput of the high-speed wireless direct connection [17,18]. Besides, there exists a non-negligible control latency between the buffer state report and sender decision-making even in our one-hop scenario (§3.3.1). Therefore, the sender can only receive infrequent feedback from the receiver and has to strategically determine a sending rate according to out-of-date buffer states. Furthermore, although our scenario only involves two entities, there are multiple buffers along the data path (§2.1). The sender thus needs to coordinate these buffers and ensure its sending rate does not cause underflow or overflow in any buffer.

In this paper, we present SMUFF, a file transfer service that improves Wi-Fi Direct transport throughput to line rate by orchestrating the on-device buffers. We model the transport data path as a series of linearly connected buffers, and consider the traffic as a fluid that sequentially traverses each buffer. Our core idea is to identify and address the bottleneck component within this data path. To maximize throughput, we maintain an appropriate backlog of data in the bottleneck buffer. The amount of backlogged data must satisfy two constraints: *i)* Enough data must be backed up to prevent a buffer underflow, *i.e.,* an empty router queue results in throughput less than the link rate; and *ii)* sufficient buffer space must be reserved to avoid buffer overflows, which can cause packet drops and complicate the subsequent packet recovery process. This leads us to define a "safe range" for the amount of buffered data. This range ensures that even in situations where state information is infrequently updated or out of date due to limited feedback, the buffer remains protected from both underflow and overflow. To accomplish this, SMUFF systematically collects state information from on-path buffers and calculates an optimal sending rate. This rate is carefully tuned to keep the bottleneck buffer size within the safe range.

SMUFF requires no proprietary hardware or modification to the device kernel, making it portable across different device models and easy to deploy. We have deployed the SMUFF service on four different Android mobile phone models with release dates spanning five years. Our evaluation shows that SMUFF achieves an average link utilization of 88.7% and 91.8% for 802.11ac and 802.11ax, respectively. It improves the link utilization by up to 22.6% while reducing CPU utilization by 37% and energy consumption by 15%, compared to the state-of-the-art solution.

Our contributions are as follows:

- We reveal the fundamental limitations of existing end-to-end transport schemes for Wi-Fi Direct. These schemes designed for multi-hop networks suffer from link under-utilization in peer-to-peer wireless data transfer.
- We design SMUFF, a file transfer service that is dedicated to Wi-Fi Direct towards practical line rate. SMUFF maximizes throughput by deriving the optimal sending rate according to the buffer states.
- We implement SMUFF to be compatible with Android and demonstrate its ability to reach up to 94.7% utilization of the wireless line rate.

This work does not raise any ethical issues.

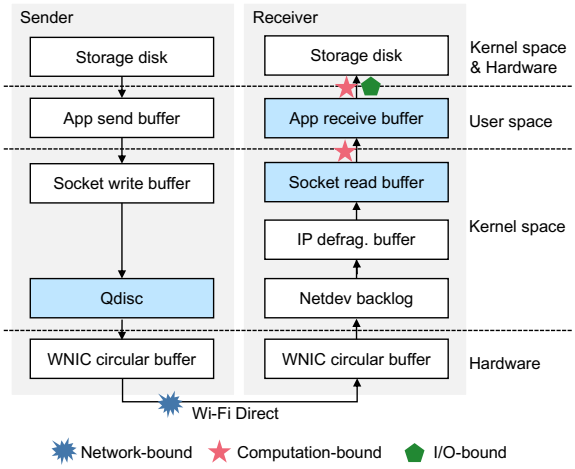

Figure 1: The end-to-end file transfer path. The Qdisc buffer, the socket read buffer, and the application receive buffer (highlighted in blue) are vulnerable to overflows.

2 Background and Motivation

In this section, we introduce the background and data packet lifecycle of Wi-Fi Direct (§2.1). We then examine the achievable wireless line rate (§2.2) and the limitations of existing transmission schemes that prevent them from achieving the line rate of peer-to-peer direct links (§2.3).

2.1 Wi-Fi Direct and Its Packet Life-Cycle

Wi-Fi Direct is a link layer (L2) technology that allows mobile devices to establish a direct Wi-Fi connection with each other directly, without the need for a central access point or any intermediate nodes. This technology is primarily used for improving daily life productivity such as file transfer between devices [20]. Note that it only provides L2 connectivity and does not specify any transport protocols or default applications. Device vendors need to choose or implement upper-layer protocols to support reliable data transfer.

The end-to-end pipeline of a file transfer task on a typical Android device (using Wi-Fi Direct) is shown in Fig. 1. To simplify the illustration and bypass the heavy TCP network stack, we use the UDP socket as an example.

Sender. The sender reads data from the disk and encapsulates it into a series of UDP datagrams. The datagrams are then sent from userspace and enqueued into the queueing discipline (Qdisc), which is a software queue that allows traffic shaping and prioritization. Note that for UDP, the socket write buffer is a virtual buffer that only counts memory allocation but has

Buffer Type	Overflow Risk Analysis
Application Send Buffer	No, the application can stop sending data when the buffer reaches its capacity.
Socket Write Buffer	No. For TCP, the kernel sets the socket as non-writable when the buffer is full. For UDP, the socket write buffer is a virtual buffer that only tracks packet memory allocation.
Qdisc	Yes, this buffer is shared among all traffic and may overflow if the Wireless Network Interface Card (WNIC) cannot drain it quickly enough.
WNIC Circular Buffer	No. At the sender, the NIC driver notifies the kernel when the firmware buffer is full. At the receiver, the kernel prioritizes the interrupt handler responsible for receiving data.
Netdev Backlog	Almost never, as the high-priority interrupt handler can process data.
IP Defragmentation Buffer	No overflow in single-hop scenarios. Overflow typically results from Denial-of-Service (DoS) attacks [19].
Socket Read Buffer	Yes, if the receiving process lacks sufficient CPU time to copy data from it.
Application Receive Buffer	Yes, if the receiving system is I/O-bound.

Table 1: Analysis of buffer overflow risk in the Wi-Fi Direct transport data path.

no real queue to buffer data. The kernel blocks the application from sending more data when the memory counter reaches a memory allocation threshold. Finally, the WNIC driver fetches packets from the Qdisc and sends them to the peer.

Receiver. The received packets are stored in the network device queue (*i.e.,* netdev backlog) by the interrupt handler and then processed by the kernel. After processing the received data, the kernel stores the UDP datagrams in the read buffer and waits for the userspace program to read them. The receiving application reads the data from the socket read buffer, performs packet reordering and loss recovery, and asks the kernel to write to disk.

Buffer Overflow Analysis. Buffer overflow is the primary cause of packet drops when we send UDP packets at a high rate. Table 1 gives an analysis of overflow risk. We need to consider these buffers when designing our system. We identify three buffers that are susceptible to overflow because of bounded device capability, as highlighted in blue in Fig. 1: *i) Qdisc*. Packet drops happen at Qdisc when the sending rate is faster than the available wireless bandwidth and the queue is full. The bounded network capacity cannot drain the queue fast enough; *ii) Socket read buffer.* When the system is under heavy load, or the processor clock frequency is constrained by the mandatory thermal throttling mechanism to avoid overheating, the receiving application process may not have sufficient CPU time slices to copy the data from the kernel. The packets could drop if the socket read buffer is full; *iii) Application receive buffer.* Like the socket read buffer, the application buffer is prone to overflow when there is no available CPU time. This buffer also overflows when the disk I/O rate cannot keep up with the receiving rate.

In a nutshell, from a buffer management perspective, we need to carefully pace the sending rate to avoid overflow in any of these buffers, preferably at the upper layer (*e.g.,* userspace transport) instead of inside the kernel or firmware in WNIC for the sake of programmability and deployability.

2.2 Practical Line Rate Transmission of Wi-Fi Direct

Theoretical transmission rate cannot be achieved in the real world. While Wi-Fi standards such as 802.11ac (Wi-Fi 5) and 802.11ax (Wi-Fi 6) claim impressive physical bandwidth limits[1] of up to 867 Mbps and 1.2 Gbps, respectively [21, 22], real-world peak throughput falls far short of these advertised rates. The gap between the line rate and the physical rate is inevitable for two reasons: *i) Protocol overhead.* Network protocols from the MAC layer to the transport layer require the process of header information in each data packet. This additional overhead reduces the effective throughput that can be achieved; *ii) Channel contention*. In areas with a high density of wireless devices or networks, Wi-Fi devices adhere to Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) mechanism [22] when accessing the wireless channel, resulting in decreased throughput [23]. To illustrate the discrepancy between the achievable transmission rate and the theoretical rate, we conduct an experiment using two 802.11ac supporting devices with a maximum physical bandwidth limit of 867 Mbps. We use the `iperf` tool [24] with fine-tuned parameters and the `iw` tool [25] to monitor the physical layer data rate reported by the wireless NIC (WNIC). As shown in Fig. 2, for the first 30 seconds we place the two devices head-to-head, allowing them to reach the maximum achievable throughput. However, although the physical layer rate consistently reaches its theoretical maximum, the actual throughput fluctuates and remains below that rate. From 30 seconds to 60 seconds, we move the devices 5 meters apart while keeping them stationary. This change caused instability in the data rate due to increased collisions and retransmissions caused by other devices sharing the same wireless channel. Between 60 seconds and 90 seconds, we deliberately introduce channel contention by running TCP traffic on additional devices operating on the same Wi-Fi channel. The actual throughput is well below the physical layer rate.

Defining practical line rate transmission. While the physical layer rate cannot serve as the metric of practical transmission rate due to protocol overhead and channel contention, previous research has explored techniques for estimating this

[1] From a wireless communication perspective, it is the instantaneous highest data rate based on the modulation and coding scheme determined by the channel condition (*e.g.,* Signal to Interference plus Noise Ratio, or SINR) and channel bandwidth according to the Wi-Fi specification.

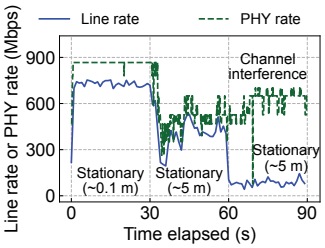

Figure 2: Traces of the physical rate and the actual UDP throughput.

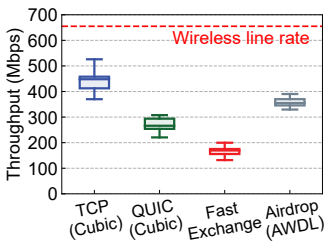

Figure 3: The Wi-Fi Direct link throughput of two 802.11ac devices 3 meters apart.

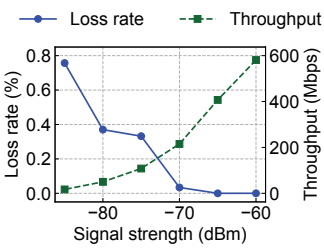

Figure 4: Wi-Fi Direct cannot guarantee packet loss recovery (at lower layer).

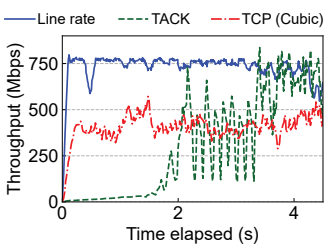

Figure 5: Congestion control fails to (quickly) achieve high utilization of a wireless link.

rate using specific hardware [15, 26]. However, these estimates are inherently approximate and hardware dependent. In the rest of this paper, we take the empirical approach in our experiment and define *practical line rate* as the maximum instantaneous throughput achievable by `iperf` UDP over a one-second window. Such a definition provides a practical and real-time measure of network performance and is a valuable metric for our analysis.

2.3 Limitations of Existing Transport Schemes

Existing peer-to-peer transmission schemes fail to achieve line rate. While the achievable line rate is below the claimed physical line rate, existing peer-to-peer transmission schemes still cannot achieve such a line rate over Wi-Fi Direct link. As shown in Fig. 3, we initiate data transmission tasks over a Wi-Fi Direct link with two transport layer schemes (TCP CUBIC and QUIC CUBIC) and two commercial services (Xiaomi Fast Exchange and Apple AWDL). While the wireless line rate is 655 Mbps in this setting, TCP (CUBIC), as the most widely used transport solution, only achieves 442 Mbps on average. QUIC and other commercial solutions perform even worse. This pilot study indicates a great opportunity to design a dedicated transport scheme for Wi-Fi Direct towards its wireless line rate. In the rest of the section, we investigate the drawbacks of basic transport layer elements in Wi-Fi Direct links and their implications on a dedicated transport solution.

Resource-Intensive Reliable Delivery Mechanism. The Medium Access Control (MAC) layer of Wi-Fi performs "best effort" retransmission upon packet loss[2]. Therefore, Wi-Fi Direct is only partially reliable, and packet loss over a wireless channel is unavoidable. We conduct a Wi-Fi Direct data transfer experiment and confirm that packet loss is not negligible when the signal is "weak" (*i.e.,* below -70 dBm), as shown in Fig. 4, which means that the reliable delivery (including loss recovery and packet reordering) is still a key requirement for Wi-Fi Direct and should be handled by the transport layer. TCP is widely used for reliable delivery, but it has the following two drawbacks when applied to Wi-Fi Direct:

• *Per-packet ACKing intensifies channel contention.* A recent study [18] shows that over a Wi-Fi link, TCP's default ACKing policy (*i.e.,* at least one ACK every other packet [28]) would hurt the throughput by 33% compared to ACKing every 16 packets. This is because uplink and downlink data in a shared wireless channel will cause extra contention and MAC protocol overhead, suggesting a reduced ACK frequency for a delicate transport layer design.

• *Computation-intensive network stack stresses computation resource.* The TCP stack is known to be computationally intensive and causes more packet processing overhead [29]. More recent transport protocols such as QUIC opt for high efficiency based on UDP, which bypasses the TCP stack and gives more flexibility to the user space. Therefore, we need a lightweight transport protocol based on UDP datagrams for its efficient in-kernel implementation.

Unsuitable Congestion Control. TCP or QUIC relies heavily on congestion control algorithms (CCAs) to adjust the sending rate. Basically, the goal of CCA is to evenly distribute bandwidth among the flows traversing a bottleneck link in a multi-hop path. However, CCAs are unsuitable in the one-hop, single-flow setting in Wi-Fi Direct links and face the following two problems:

• *Unnecessary Startup Phase.* CCAs need to probe the available bandwidth and send packets conservatively to avoid congestion, adding unnecessary start time to quickly reach the peak sending rate. Our experiment in Fig. 5 shows that Cubic spends 0.15 seconds on the slow start phase, while TACK [18], a TCP variant that reduces ACK frequency to alleviate channel contention, spends 3.3 seconds on the congestion avoidance phase. In the peer-to-peer scenario, the slow start should be avoided because of the absence of congestion.

• *Overreact to Lossy Link.* CCAs could reduce the congestion windows to lower the sending rate due to packet loss or transient delay spikes, which are common on wireless links. This overreaction leads to link under-utilization. As shown in Fig. 5, both CCAs cannot reach the available link bandwidth and the throughput oscillates due to wireless link fluctuation. A wireless transport layer should be insensitive to a lossy link.

Inefficient Flow Control. Flow control is designed to prevent the sender from overwhelming the receiver. To achieve line

[2]It retries for failed transmission until the attempt reaches `dot11ShortRetryLimit` (7 by default) or the retransmission time exceeds `dot11MaxTransmitMSDULifetime` (512 ms by default) [27].

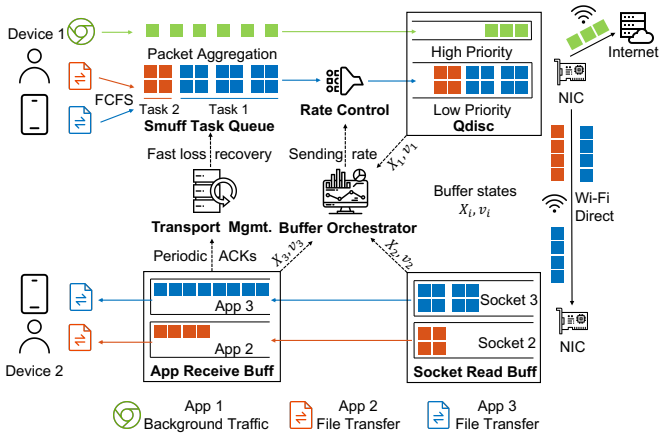

Figure 6: SMUFF Overview.

rate transmission in Wi-Fi Direct, more delicate flow control should be used because *i)* the sender needs to decide a proper packet sending rate to achieve line rate while avoiding packet drops. An insufficient sending rate will not fully utilize the WNIC potential, causing link under-utilization, while an excessive sending rate may cause packet drops at the sender Qdisc due to the limited queue length. *ii)* With Gigabit wireless hardware equipped, the receiver could fall short of handling such a high speed due to CPU and I/O bottlenecks and incur buffer overflow. The existing flow control in TCP and QUIC is inefficient in the following three aspects:

• *Unaware of other On-path Buffers.* Conventional TCP flow control only considers transport-layer buffers and is agnostic to other buffers in the packet life-cycle – underflow or overflow on any of them can still cause link under-utilization – a new flow control design should take all buffers into account.

• *Delayed ACK Feedback.* The flow control window in TCP and QUIC assumes ACK-driven updates, which may be delayed due to a longer packet queue or ACK frequency reduction [18]. The window may be slow to adjust to the fluctuating wireless link due to delayed feedback. The effectiveness of flow control should not depend on timely ACKs.

• *Fixed Parameter Configuration.* The maximum flow control window size is a fixed value during transmission [30, 31] that varies between implementations and device configurations. Such a value will not be ideal for other buffers of different sizes in the packet lifecycle and different link capacities. Flow control should better handle all of this heterogeneity, dynamics, and complexity.

3 SMUFF Design

In this section, we propose SMUFF, a Wi-Fi Direct file transfer service to achieve line rate by orchestrating on-device buffers. We first present the design goals of SMUFF.

3.1 Design Goals

The main idea of SMUFF is to orchestrate multiple buffers by actively monitoring on-device buffer states and continuously obtaining an optimal sending rate. We design SMUFF to achieve the following three goals.

G1: Link Bandwidth Utilization Maximization. As presented in §2.3, existing transport solutions often fall short of achieving line rate transmission. SMUFF should be designed to make the full utilization of available Wi-Fi direct link bandwidth by dynamically adapting its sending rate at the transport layer to saturate the network interface.

G2: On-path Buffer Overflow Avoidance. A straightforward idea is to continuously send packets to saturate the link. However, this may cause unnecessary packet loss due to buffer overflow and complicate the loss recovery process. SMUFF should effectively manage the critical packet buffers along the data path by monitoring the buffer states and making timely adjustments to the sending rate to avoid packet loss.

G3: Practical Deployment on Mobile Devices. SMUFF is designed for easy portability to different phone models. To make SMUFF practical for deployment, it should not rely on vendor-specific hardware (*e.g.,* proprietary NICs). In addition, it must have a low CPU usage footprint and minimize its impact on concurrent flows on the same devices.

3.2 System Overview

SMUFF is designed as a system service, and an overview of its workflow and main system components are shown in Fig. 6. This service provides a platform for applications on the sender side to submit file transfer tasks, which are then managed and executed by SMUFF on a first-come, first-served (FCFS) basis. SMUFF needs to efficiently transfer the files to the corresponding application on the receiving device.

In the rest of this section, we first present our theoretical analysis to achieve line rate data delivery (§3.3). We then put our analysis into practice with two key system components: *Buffer orchestrator* (§3.4) collects buffer states and calculates an optimal sending rate to maximize system throughput; *Transport manager* (§3.5) ensures reliable data delivery and facilitates packet loss recovery.

3.3 Buffer Management Analysis

As shown in §2.1, data is repeatedly processed and transferred to the next buffer until it reaches its final destination. This property allows us to take a buffer-by-buffer approach for flow management. For ease of exposition, we will start with the management of a single buffer (§3.3.1) and then extend the methodology to multiple buffers (§3.3.2).

3.3.1 Single Buffer Management

To derive the optimal sending strategy, we begin with the simple case where there is only one buffer as shown in Fig. 7. The packet buffer stores the incoming data until the data can be processed by the next system component. We suppose the maximum size of the buffer is $X_{\max}$. We denote the packet ingress and egress rate as v_{in} and v_{out} respectively and denote the current amount of data in the buffer as X. The main purpose of the buffer is to serve as a cushion to absorb the

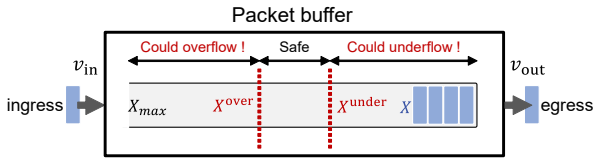

Figure 7: Single buffer management.

variation in the ingress and egress rate. In order to make full use of the buffer, we translate the design goals G1 and G2 into the following two constraints.

C1: Backlog enough data to avoid buffer underflow. To keep the system working at its full rate, the buffer needs to be backlogged (*i.e.,* the buffer never runs out of data). If the buffer goes empty, the egress rate will decrease and fail to achieve the line rate.

C2: Reserve enough buffer room to avoid buffer overflow. To prevent data loss, it is also necessary to allocate sufficient buffer space to handle incoming data. A buffer overflow occurs when the data pushed into the buffer exceeds its capacity, resulting in potential loss of information and disruption of system functionality.

Challenge Brought by Control Latency. Both C1 and C2 can be easily satisfied if we are able to control the ingress rate v_i just in time. We can increase v_i when the buffer is about to underflow and decrease it when the buffer is about to overflow. However, the conventional assumption of low-latency data delivery in single-hop wireless networks does not hold for the line rate transmission. There is a non-negligible *control latency* in buffer management, which we define as the time interval between buffer state reporting and the rate adjustment taking effect. For example, the buffer states at the receiver must be relayed to the sender, and the control latency is dominated by the RTT of the data flow. To demonstrate such latency, we initiate TCP connections with an increasing sending rate over an 802.11ac Wi-Fi direct link and obtain RTT from the kernel data structure. As shown in Fig. 8, the RTT increases up to 25 ms with the increase in the system throughput even in this single-hop setting. This intriguing phenomenon finds its roots in the well-documented bufferbloat issue [32]. Notably, this issue becomes increasingly pronounced as throughput scales to larger proportions, contrasting the experiment result in previous work [15,33,34] where the latency is no more than a few mill-seconds due to a low throughput of much less than 20 Mbps. Therefore, in the presence of the control latency, we need to strategically select the ingress rate v_i to maintain the right amount of data in the buffer.

Buffer Occupancy Estimation. The change in buffer occupancy X depends on the delta of the ingress rate and the egress rate. This can be formulated as:

$$\dot{X} = v_{\text{in}} - v_{\text{out}} \tag{1}$$

where $\dot{X}$ is the derivative over the time t (*i.e.,* $\dot{X} = \frac{\mathrm{d}}{\mathrm{d}t}X$). Suppose at time t_0, the buffer reports its current states, including $X(t_0)$, $v_{\text{in}}(t_0)$, and $v_{\text{out}}(t_0)$. The rate adjustment takes effect at a later time $t_1 = t_0 + d$ with a control delay d. Therefore, we need to estimate the buffer occupancy during t_0 to t_1 based on the reported values to adjust the sending rate at t_1. We make estimations by exploiting the fact that there exists a lower and an upper bound for the processing rate of any system components. For example, the wireless transport rate must be greater than zero and less than the physical limit. Therefore, we can assume a value range for $v_{\text{in}}(t)$ and $v_{\text{out}}(t)$ for the duration from t to $t+d$:

$$v_{\text{in}}^{\text{lo}}(t) \le v_{\text{in}}(t) \le v_{\text{in}}^{\text{up}}(t) \tag{2}$$

$$v_{\text{out}}^{\text{lo}}(t) \le v_{\text{out}}(t) \le v_{\text{out}}^{\text{up}}(t) \tag{3}$$

As a result, we can infer the value range of $\dot{X}$ by Eqn. 1, Eqn. 2, and Eqn. 3:

$$v_{\text{in}}^{\text{lo}}(t) - v_{\text{out}}^{\text{up}}(t) \le \dot{X} \le v_{\text{in}}^{\text{up}}(t) - v_{\text{out}}^{\text{lo}}(t) \tag{4}$$

In order to satisfy C1, we must have a sufficient amount of data in the buffer. From Eqn. 4 we know that from time t_0 to t_1, the maximum amount of data that can be consumed from the buffer is

$$X^{\text{under}} = -d \times (v_{\text{in}}^{\text{lo}}(t_0) - v_{\text{out}}^{\text{up}}(t_0)) \tag{5}$$

As a result, if the data in the buffer is less than X^{under} at t_0, the buffer may run out of data, resulting in underutilization.

In order to satisfy C2, we need to reserve enough room in the buffer. Similarly, the maximum amount of data that can be backlogged from t_0 to t_1 is $d \times (v_{\text{in}}^{\text{up}}(t) - v_{\text{out}}^{\text{lo}}(t))$. Therefore, to avoid overflow and packet loss, the data in the buffer at t_0 should be kept below a threshold

$$X^{\text{over}} = X_{\text{max}} - d \times (v_{\text{in}}^{\text{up}}(t_0) - v_{\text{out}}^{\text{lo}}(t_0)) \tag{6}$$

To satisfy both constraints, the buffer occupancy X has a "safe" range between two thresholds X^{under} and X^{over} at t_0, as shown in Fig. 7. If X falls within this safe range, we can ensure that the next system component will consistently operate at full speed and that no packet loss will occur for the next time period of control latency d. The existence of this safe region depends on $X^{\text{under}} < X^{\text{over}}$, meaning that the buffer size X_{max} must exceed a certain threshold, specifically $d \times (v_{\text{out}}^{\text{up}}(t_0) - v_{\text{in}}^{\text{lo}}(t_0) + v_{\text{in}}^{\text{up}}(t_0) - v_{\text{out}}^{\text{lo}}(t_0))$.

3.3.2 *Multiple Buffer Management*

There are typically multiple buffers (or queues) connecting different system components in a transport system. The data starts from the sender, gets processed and traverses each component and buffer, and arrives at the receiver. In our case as shown in Fig. 9, the Qdisc at the sender-side stores the packets sent by the application and waits for the network interface driver to fetch data to send. Also, the application buffer stores the data that is read from the transport layer and waits for the I/O requests to be ready.

While there are three buffers in our Wi-Fi direct use case, this idea can be generalized to more buffers. Without loss of generality, suppose there are n buffers. We let X_i denote the buffer occupancy (*i.e.,* buffer size or queue length) of the i-th

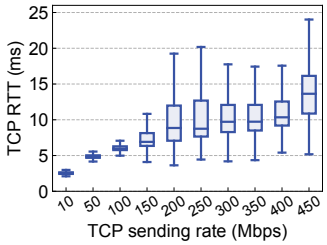

Figure 8: TCP control latency (RTT) increases as system throughput increases.

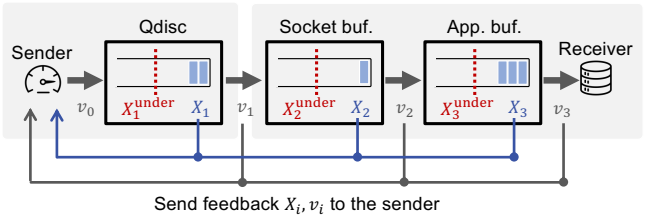

Figure 9: Multiple buffer management for the transport over the Wi-Fi direct connection.

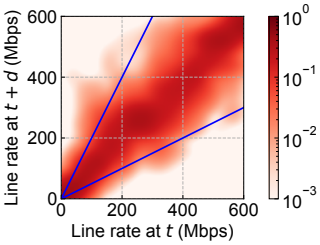

Figure 10: Throughput transition probability within a control latency ($d = 40$ ms).

buffer. We let v_i denote the instantaneous data rate from the i-th buffer to the $(i+1)$-th buffer. Note that v_0 and v_n are the sending and receiving rates, respectively.

Our goal is to adjust the sending rate to maximize the overall throughput of the system. While the overall throughput is the processing rate of the bottleneck system component, our idea is to identify the bottleneck component and maintain a moderate backlog of data in the buffer connected to that particular component. Our algorithm to determine the sending rate v_0 works in two steps: *i)* calculate the target amount of data to backlog for each buffer; *ii)* adjust the sending rate to maintain enough data at the bottleneck buffer.

Calculating the target backlog size. We need to ensure that the backlog size remains within the safe range at all times. To minimize the buffer usage, we set the *target backlog size* X_i^{target} for the i-th buffer to the lower bound of the safe range (X_i^{under}). On each latest feedback signal (v_i, X_i) from the i-th buffer, we update X_i^{target} according to Eqn. 5 as follows:

$$X_i^{\text{target}} = -d_i \times \begin{cases} (v_{i-1}^{\text{lo}} - v_i^{\text{up}}) & \text{if } i > 1 \\ -v_i^{\text{up}} & \text{if } i = 1 \end{cases}, \tag{7}$$

where d_i is the control latency for the i-th buffer, $v_i^{\text{lo}}, v_i^{\text{up}}$ are the lower bound and upper bound of v_i. We leave the settings of $v_i^{\text{lo}}, v_i^{\text{up}}$, and d_i in practice to §3.4. Note that we can also set the target backlog size X_i^{target} to any other value within the safe range (for example, the midpoint of the safe range $(X_i^{\text{under}} + X_i^{\text{over}})/2$).

Calculating the sending rate. For each feedback signal, the sender determines the sending rate based on the following:

$$v_0 \leftarrow \max\left(\min_{1 \le i \le n}\left\{v_i + \frac{X_i^{\text{target}} - X_i}{d_i}\right\}, 0\right), \tag{8}$$

The interpretation of this rule is as follows. For the i-th buffer, when the current buffer occupancy is less than the target buffer backlog size ($X_i < X_i^{\text{target}}$), this buffer could underflow, resulting in link underutilization. Therefore, we need to increase the rate v_i to increase the buffer occupancy in a control delay time d_i. Similarly, when the buffer occupancy exceeds the target backlog size ($X_i > X_i^{\text{target}}$), we decrease the sending rate. The outer min operator is used to identify the bottleneck of the whole system.

3.4 Buffer Orchestrator

The buffer orchestrator bridges the gap between theoretical buffer management analysis (§3.3) and practical implementation. Effective buffer management depends on the ability to quickly and accurately estimate system conditions. Therefore, the buffer orchestrator needs to perform two tasks: continuously monitoring buffer states and dynamically adjusting the sending rate.

Monitoring the Buffer States. There are four variables that characterize a buffer state.

- Buffer Occupancy (X): This variable indicates the current amount of data in the buffer and can be obtained directly by reading from the buffer.
- Data Consumption Rate (v): This variable represents the rate at which data is being consumed from the buffer. It is estimated over the duration of the control delay and further smoothed using an Exponentially Weighted Moving Average (EWMA) filter.
- Lower and Upper Bounds of the Data Rate (v^{lo} and v^{up}): These two variables specify the range of values for the data rate. Different buffers may have distinct properties, which we discuss as follows.

• *Bounds of the network data rate.* We conduct an empirical study to observe throughput variations. Our experiment involves two mobile devices running 802.11ac; the link is capable of a maximum PHY rate of 867 Mbps. The devices are held by two users moving randomly around a room. To saturate the link, we run an iperf UDP test while fine-tuning the iperf parameters to ensure that the maximum throughput is achievable. The result, shown in Fig. 10, is presented in a heatmap format and plot the transition probabilities from data rate at time t to rate at time $t+d$, where d is the control latency. Two reference lines in blue, $y = \frac{1}{2}x$ and $y = 2x$, are included to denote throughput reductions of $0.5\times$ and increases of $2\times$, respectively. Our results show that within a control latency duration, the network throughput is highly unlikely to experience a $> 2\times$ increase or a $< 0.5\times$ decrease, with a probability greater than 99%. Therefore, for the Qdisc, we set the lower bound $v_1^{\text{lo}} = 0.5v_1$ and upper bound $v_1^{\text{up}} = 2v_1$. Note that this empirical measurement is specific to our setup.

The throughput variation is affected by the complex wireless channel environment. To completely prevent buffer overflows due to rapidly changing channel conditions, SMUFF can use strict bounds on the data rate, setting the lower bound to zero and the upper bound to the maximum theoretical data rate.

• *Bounds for CPU processing and I/O rate.* To determine the bounds of the processing rate of the socket buffer and the application buffer at the receiver, we take advantage of the insight that these rates are largely unaffected by the network, but are CPU- and I/O-bound (Fig. 1). Therefore, they tend to exhibit stability within a device that is largely dependent on the underlying hardware configuration. As a result, for the socket buffer and application buffer, we set the lower and upper bounds to the values of 0.9 and 1.1 times the estimated data rate, respectively, to allow for error tolerance.

Controlling the Sending Rate. In response to signals indicating buffer state updates, the buffer orchestrator calculates the new sending rate using the formula in Eqn. 8 and adjusts v_0. While the sending rate is determined at the sender, the control latency d_i varies depending on the location of the buffers: *i)* When dealing with the Qdisc buffer at the sender, the control latency can be set to a relatively small value (4 ms in our implementation), because polling the buffer state information from the sender buffers can be done simply by a system call; *ii)* When dealing with the two buffers at the receiver, the control latency is set to the network latency.

3.5 Transport Manager

While buffer orchestrator carefully monitors buffer states and adjusts the sending rate accordingly, packet loss could still occur due to lossy wireless link. The loss detector guarantees data delivery reliability based on the sliding window mechanism and ACKing with two improvements.

Reduce ACK Frequency. Frequent ACK messages exaggerate channel contention and have a negative impact on throughput in Wi-Fi direct transport (§2.3). To address this issue, we use periodic ACKs and send four ACKs per RTT to reduce the ACK frequency. This setting proves to be robust in practice [18]. In addition, an ACK message also carries the states of the buffers at the receiver side.

Speed up Loss Recovery. We propose three optimizations to improve the efficiency of the loss recovery procedure. *i)* The use of periodic ACKs prevents the receiver from notifying the sender of lost events until the next ACK is sent. To speed up recovery, SMUFF immediately sends an ACK containing a retransmission request when a loss event is detected. *ii)* To reduce overhead, we use Negative Acknowledgments (NACKs), similar to the SACK option [35] in TCP and the ACK range in QUIC [31], to inform the sender of missing data. *iii)* We use PING messages to periodically probe the RTT and use an EWMA filter to obtain a smoothed RTT estimate. This design mitigates RTT measurement bias caused by reduced ACK frequency and does not require additional computation or maintenance of a complex per-packet data structure.

3.6 Other Design Considerations

In practice, network-tuning is also important to achieve high throughput [29, 36–38]. Here we highlight two important considerations in the SMUFF design.

3.6.1 Packet Aggregation

Packet aggregation can reduce processing overhead in software [29]. SMUFF uses packet aggregation to improve throughput and reduce mobile computation and power costs. Specifically, SMUFF generates full-size UDP packets, approaching 64 KB in size, instead of the usual 1500-byte packets. The IP layer then fragments or reassembles these packets to meet the Maximum Transmission Unit (MTU) limit, conforming to the IP fragmentation feature [39]. Since a substantial amount of energy is drained by the processing of packet units (*i.e.,* independent of their size, air time, or modulation and coding scheme) [40], SMUFF is more computation- and energy-efficient with packet aggregation. The effectiveness of packet aggregation is evaluated in §5.4.

However, packet aggregation can lead to increased susceptibility to packet loss. In cases where a single packet within a fragmented small packet is lost, the entire IP packet is considered lost. To illustrate this concept mathematically, if the network path loss rate is denoted as L, then the packet loss rate for a full-sized IP packet can be expressed as $1-(1-L)^n$, where n is the number of fragments into which the IP packet is broken. For example, if the path loss rate is 1% and a full-size UDP packet is segmented into 43 fragments, the total packet loss rate rises to an unacceptable 35.1%, significantly reducing the available throughput. To address this issue, we consider the infrequency of packet loss events in most usage scenarios, a phenomenon that contrasts with Internet connections where loss is often attributed to network congestion. Therefore, we empirically set a conservative threshold of 0.1%. Packet aggregation is only enabled if the current network loss rate is below this threshold. We provide a evaluation of its performance and the rationale for choosing this threshold in the evaluation (§5.3).

3.6.2 Flow Prioritization

Data transfer is often a background job. Other foreground traffic flows are potentially at a disadvantage when sharing the same outbound device with SMUFF because the buffer orchestrator will try to keep the Qdisc blocklogged (if the network is the bottleneck). This can affect interactive traffic such as web browsing and video streaming. To address this issue, we strictly prioritize other foreground traffic over the SMUFF flow. Specifically, there are multiple queues with different priorities at the Qdisc layer, and the packet in the queue with the higher priority is sent first. Therefore, by setting a lower priority for the SMUFF packets, we can reduce the impact on other traffic. The effectiveness of flow prioritization is evaluated in §5.3.

4 Implementation

We have developed SMUFF for commodity Android devices as a dedicated system daemon running entirely in userspace, with a primary focus on minimizing development and integration complexity. Our implementation consists of about 8000 lines of C++ code and seamlessly integrates with the NDK toolchain as well as the native dependencies inherent to the Android platform. SMUFF extends its functionality by providing a file transfer service over a Linux socket. Applications wishing to use this service can effectively communicate their file transfer requirements by interacting with this socket. We use the `SO_PRIORITY` option to define the priority for all packets to be sent by SMUFF. In addition, we set the maximum buffer size $X_{\max}$ to twice the bandwidth-delay product (BDP) to ensure that the safe area exists.

In SMUFF, we use standard system APIs to retrieve buffer state information. Specifically, to retrieve Qdisc information, we use the `RTM_GETQDISC` option of the Linux routing socket. The Linux routing socket is part of Netlink, which allows userspace programs to communicate with the kernel via a socket interface. To get the length of the Qdisc queue, we first create a socket with the domain `AF_NETLINK` and the protocol `NETLINK_ROUTE`. After sending the request to the socket, the kernel takes over the processing of the request and places the result in the socket receive buffer. We then read from the socket and get the length of the Qdisc queue. To get the state of the receive buffer, we use the socket option `SO_MEMINFO` to get memory-related information. It's worth noting that all APIs used are available after Linux 4.12 and have been supported since Android 9. This compatibility ensures a high degree of compatibility; we verify that SMUFF works on at least four different phone models without any code changes.

5 Evaluation

Testbed. We use two Pixel 4 devices for most of the experiment because Android is based on Linux and more readily customizable. The devices support up to 2 MIMO spatial streams, 80 MHz bandwidth, short guard interval, and default rate adaptation, enabling speeds of up to 867 Mbps. To ensure the robustness of our results, each test is repeated for 30 times, guaranteeing statistical reliability and comprehensive data collection. The experiment is conducted in a public office with over 10 Wi-Fi APs. This experiment environment closely mirrors common real-world scenarios involving direct device-to-device connections.

Baselines. We compare SMUFF with the following transport schemes as baselines:

- TACK [18, 41] is a variant of TCP. It aims to improve wireless transport performance by minimizing the ACK frequency to reduce wireless channel contention.
- CUBIC [13] uses the standard TCP socket to send the data. To make sure it gets the maximum throughput, we set the socket buffer size to 20 MB and enable the TCP window size scaling [42, 43].
- BBR [12] uses the standard TCP with BBR congestion control algorithm. As BBR currently does not ship off-the-shelf with the phones, we recompile the Android kernel to include BBR and flash the new kernel.
- QUIC [31, 44] is designed to improve transport performance for HTTPS traffic. It is based on UDP and is implemented entirely in userspace as SMUFF.
- UDP with the default system configuration is also evaluated. Note that it does not provide reliable delivery.

Signal Strength	-20 dBm	-40 dBm	-60 dBm	-80 dBm
Distance	$<$ 0.1 m	2~3 m	8~10 m	N/A
Scenario	head-to-head	face-to-face	meeting	behind walls

Table 2: Wireless signal strength for typical scenarios.

5.1 Transport Performance

Throughput. We begin by evaluating SMUFF for different levels of signal strength. The typical conditions and common usage scenarios are shown in Table 2 for reference. The average throughput of different transport schemes transferring a 1 GB file is illustrated in Fig. 11. *For good Wi-Fi signal* (≥ -60 dmB), we maintain control over signal strength by manipulating the distance between the mobile phones, thus simulating different channel environments relevant to file transfer scenarios. In all cases, SMUFF outperforms other transport schemes. Across the signal strength settings, it achieves throughput improvement from 17.8% to 18.2% compared to TACK, the state-of-the-art variant designed for wireless networks, and a remarkable 22.6% to 44.5% improvement compared to CUBIC. In terms of throughput variance, SMUFF exhibits minimal variation across different signal strengths, ensuring a consistent and reliable transmission service. This stability is due to SMUFF's ability to effectively manage data within buffers. BBR also exhibits low variance, but it achieves lower throughput due to its goal to achieve the minimum `RTprop` time. As saturating the link can result in increased latency, BBR is conservative in increasing its sending rate. In contrast, other transport schemes such as TACK, CUBIC, and QUIC, exhibit significant throughput variance as they struggle to accurately measure wireless bandwidth. Under conditions of *bad Wi-Fi signal*, where the signal strength is at -80 dBm, all solutions perform similarly, with SMUFF still approaching the practical line rate. Overall, our results consistently demonstrate the superior network throughput performance of SMUFF across different signal strength scenarios, confirming its effectiveness as a reliable data transfer solution.

Flow Completion Time. We compare flow completion times (FCT) when transferring files of different sizes, as shown in Fig. 12. For smaller files (*i.e.,* 10 MB), the FCT is mainly influenced by the time taken to reach the maximum link rate. SMUFF's ability to bypass the slow start process leads to remarkable reductions in transfer times, ranging from 55.7% to 91.4% compared to alternative solutions. In particular, TCP

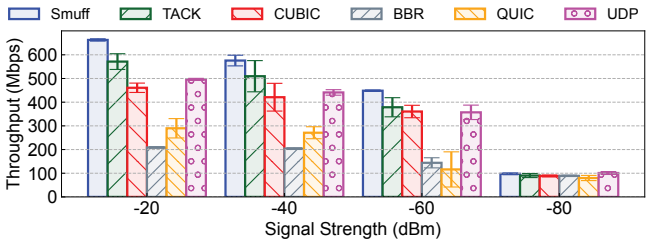

Figure 11: Throughput of different transport schemes at different signal strengths.

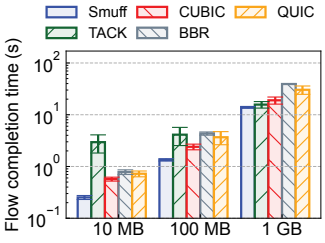

Figure 12: FCT of different transport schemes.

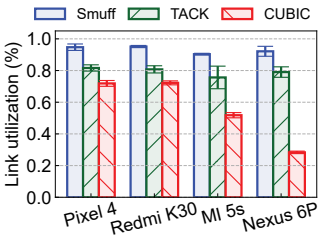

Figure 13: Link utilization on different device models.

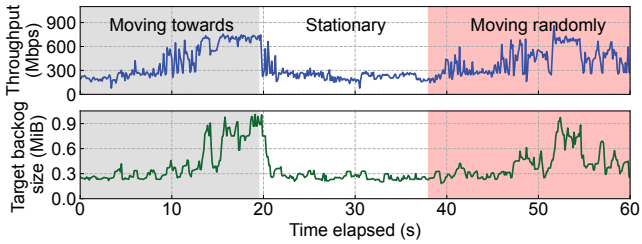

Figure 14: Trace of the real-time throughput and the target backlog size of the Qdisc.

variants and QUIC relying on the conventional slow start mechanism exhibit longer transfer times. TACK, on the other hand, shows significant variance and a much longer FCT compared to all other schemes, mainly due to its extended slow start phase caused by ACK shortages. This result is consistent with the analysis presented in §2.3. For medium-sized files (*i.e.,* 100 MB), SMUFF remains the best scheme with a 44.2% to 67.4% reduction in FCT over the other schemes. For larger files (*i.e.,* 1 GB), FCT is mainly determined by the link utilization. Consequently, TACK shows better performance compared to transferring smaller files. SMUFF still outperforms TACK by reducing the FCT by 12.1%. Furthermore, when averaged across CUBIC, BBR, and QUIC, SMUFF reduces transfer time by an impressive 26.3% to 64.4%. This experiment shows that SMUFF can significantly reduce the FCT by fully utilizing the link capacity.

Robustness to Mobility. As discussed in §3.3.2, the buffer orchestrator continuously computes the target backlog size based on newly received buffer states. To evaluate how effectively SMUFF adapts to mobility, we perform mobility tests focusing on the target backlog size of the Qdisc buffer. Our experiment involves two users with devices moving around a room, simulating realistic mobility scenarios. The real-time throughput collected at the receiver and the target backlog size of the Qdisc are recorded, as shown in Fig. 14. The experiment includes three different movement patterns: For the first 19 seconds (the gray region), both users move toward each other. Between 19 seconds and 38 seconds (the white region), both users move away from each other and then remain stationary. After 38 seconds (the red region), the users roam randomly in the room. The results show that the target backlog size closely matches the throughput. Such synchronization demonstrates SMUFF's ability to make robust adjustments to its sending rate to prevent overflow or underflow under mobility. This adaptability to mobility ensures stable and reliable performance in dynamic network environments.

	Redmi K30 Pro	Google Pixel 4	MI 5s	Nexus 6P
SoC	Snapdragon 865	Snapdragon 855	Snapdragon 821	Snapdragon 810
RAM	6 GB	4 GB	3 GB	3 GB
Battery	4700 mAh	2800 mAh	3200 mAh	3450 mAh
Wi-Fi	802.11a/b/g/n/ac/ax	802.11a/b/g/n/ac	802.11a/b/g/n/ac	802.11a/b/g/n/ac
Released	2020	2019	2016	2015

Table 3: Device specifications.

5.2 Compatibility

Different Device Models. To demonstrate the deployability and adaptability of SMUFF, we conduct deploy SMUFF and performance tests on four different smartphone models listed in Table 3. We compare SMUFF with two sub-optimal schemes (TACK and CUBIC) and initiate our tests by designating each device as the sender, with the Pixel 4 serving as the receiver. This configuration was chosen because the sender actively manages the sending rate based on buffer states, providing a robust metric. The results, shown in Fig. 13, show that SMUFF outperforms TACK and CUBIC on all devices, achieving a remarkable link utilization rate (achieved throughput divided by line rate) of up to 94.7%. It's worth noting that CUBIC's performance shows some variability across devices, likely due to differences in processing capabilities. Overall, our experiments confirm that SMUFF works seamlessly across different device models, delivering consistently high link utilization.

Wi-Fi Specifications. To evaluate link utilization with different Wi-Fi specifications, we conduct experiments using a stable signal with a strength of -40 dBm. This evaluation is performed on both 802.11ac (Wi-Fi 5) and 802.11ax (Wi-Fi 6) networks. We evaluate the throughput of SMUFF against all other transport schemes. The results, shown in Fig. 15, highlight the impressive performance of SMUFF, which is consistently close to the line rate. It achieves an average link utilization of 88.7% on 802.11ac (equivalent to 576 Mbps) and an even more remarkable 91.8% on 802.11ax (equivalent to 645 Mbps). These results demonstrate SMUFF's ability to efficiently utilize available network resources across different Wi-Fi specifications.

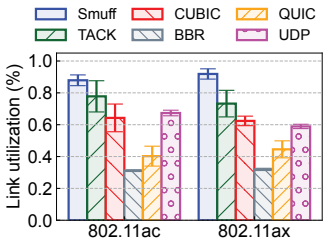

Figure 15: Link utilization of different transport schemes and Wi-Fi specifications.

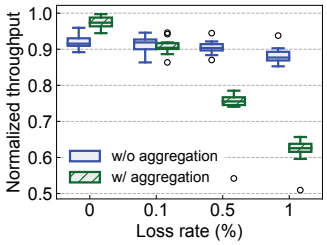

Figure 16: Impact of loss rate on the packet aggregation.

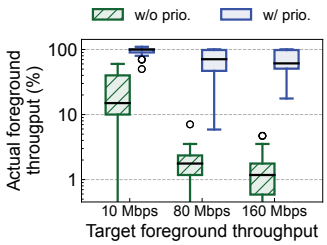

Figure 17: Flow prioritization improves the foreground throughput.

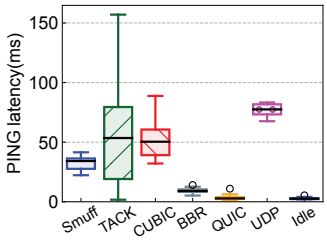

Figure 18: PING latency of different transport schemes.

5.3 Microbenchmark

We perform some micro-benchmarks to investigate the impacts of design choices made in the development of SMUFF.

Packet Aggregation Robustness. As discussed in §3.6.1, packet aggregation can make a network more susceptible to packet loss. To investigate how SMUFF behaves under various conditions, we perform tests at -40 dBm signal strength, both with and without packet aggregation. To simulate packet loss, we drop packets randomly at the receiver. As shown in Fig. 16, in scenarios where no packet loss occurs, packet aggregation effectively reduces processing overhead and improves performance, resulting in an average improvement of 5.4%. However, as the loss rate escalates from 0.1% to 1%, throughput drops dramatically by up to 62.4%. Such a severe reduction is primarily due to the vulnerability introduced by packet aggregation, where the loss of a single IP fragment packet results in the loss of the entire packet. In contrast, when packet aggregation is disabled, SMUFF shows efficiency in recovering lost packets. Therefore, SMUFF disables aggregation when the path loss rate exceeds a predefined threshold, ensuring reliability and preserving performance in the presence of a high packet loss rate.

Flow Prioritization. As discussed in §3.6.2, SMUFF can potentially cause foreground traffic to starve. To evaluate the effectiveness of flow prioritization in mitigating this problem, we perform tests by running foreground TCP flows alongside SMUFF. The TCP flows are configured with different target throughputs (10, 80, and 160 Mbps). We measure the actual throughput achieved by the TCP flows with flow prioritization enabled and disabled. As shown in Fig. 17, without flow prioritization, the foreground TCP flows achieve an average throughput of 2.44 Mbps, 2.74 Mbps, and 2.54 Mbps for different target throughputs, indicating a disadvantaged state. With flow prioritization enabled, the achieved throughput improves significantly, reaching averages of 2.44 Mbps, 59.1 Mbps, and 118.2 Mbps for the respective target throughputs. While SMUFF cannot completely eliminate the impact on other flows (*e.g.,* only 73.9% of the target throughput is achieved for the 160 Mbps flow) due to the lack of prioritization in the device firmware buffers, flow prioritization significantly alleviates the problem of foreground traffic starvation.

	SMUFF	TACK	CUBIC	BBR	QUIC	UDP	Idle
Energy (J)	50.7	62.6	59.6	86.6	72.8	55.2	40.6
Overhead (%)	24.8	54.2	48.2	113.3	74.4	36.0	N/A

Table 4: Average energy consumption for transferring a 1 GB file via peer-to-peer wireless data transfer. "Overhead" is the additional energy consumption compared with the idle state.

5.4 System Overhead

Computation Overhead. To compare the computational overhead of SMUFF, we transfer a 1 GB file using SMUFF and other transport schemes and measure their CPU usage. As shown in Fig. 19, SMUFF achieves the lowest CPU usage. There are two reasons for this: First, SMUFF achieves nearly the wireless line rate and completes the transmission process much faster; second, SMUFF is based on UDP and thus avoids the overhead of a compute-intensive TCP stack. We also evaluate the effectiveness of packet aggregation. As shown in Fig. 20, aggregation can reduce the CPU usage of SMUFF by 18.0% and 21.9% on the sender and receiver side, respectively. This result shows that SMUFF can reduce system load on resource-constrained mobile devices.

Energy Overhead. We also examine power consumption, since mobile devices are typically powered by limited batteries. We use the Monsoon power monitor [45] to measure the average power consumption while transferring a 1 GB file using SMUFF and other transport schemes. The monitor works by precisely measuring the input power of the smartphone with a sampling rate of 5000 Hz and is widely used in the research community for measuring power consumption [46, 47]. As shown in Table 4, SMUFF consumes less energy than all other schemes (24% overhead over baseline) because it bypasses the computationally intensive TCP stack and completes the transmission earlier. The result shows that SMUFF is more energy efficient than other solutions.

Transmission Latency. As buffer saturation may increase transmission latency for all data flows between peer devices, we also evaluate the impact of the transport schemes on transmission latency. To measure the latency overhead of SMUFF, We conduct experiments by sending the ICMP PING messages [48] every second in the background during the file transmission. As shown in Fig. 18, compared with TACK

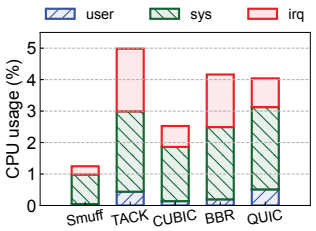

Figure 19: CPU usage of different transport schemes.

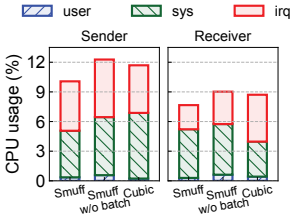

Figure 20: CPU usage reduction of packet aggregation.

and TCP CUBIC, SMUFF reduces average latency by 36.9% and 39.2% because it uses an adaptive adjustment on the backlogged data at Qdisc buffer. BBR and QUIC maintain a low latency but fail to fully utilize the available link. SMUFF keeps the latency below 40 ms with a small variance – such an increase in latency is acceptable in file transfer scenarios.

6 Discussion

Operating in Non-Wi-Fi One-Hop Networks. SMUFF does not require the underlying NIC to be a Wi-Fi device. Therefore, SMUFF could be seamlessly deployed over any NICs that consume packets from Qdisc. The effectiveness of SMUFF is based on two premises: *i)* Sending at line rate would generate the best performance. *ii)* NIC itself has some mechanism like BQL [49] to minimize the underlying buffering. Such conditions hold for most cases. Therefore, SMUFF theoretically also works in other one-hop networks such as Apple Wireless Direct Link (AWDL) [2], Neighbor Awareness Networking (NAN) [50], and wired links.

Benefiting Other Applications. SMUFF can also be used for other data transfer applications, such as 360-degree video streaming. SMUFF can introduce a minimal initial delay due to its extended queue length, but allows for higher video resolution due to its increased throughput once the video begins. To minimize latency, SMUFF can also be configured to maintain a near-zero queue length by setting the target queue size to zero. This strategy effectively reduces queue latency within the monitored buffers. However, it is important to note that latency cannot be similarly reduced for buffers that are not directly monitored, such as the NIC firmware buffer.

7 Related Work

Queue-based Transport Improvement. The packet queue provides lower-layer information to the transport control algorithm. This information enables the transport layer to perform better sending rate adjustment and accomplish different optimization goals. XCP [51], RCP [52], and ABC [15] leverage the dequeuing rate and the current queue length to let the network middleboxes compute the right sending rate and signal this rate to the sender. The signal lets the sender "jump" to the correct sending rate to achieve high throughput. Active Queue Management (AQM) schemes such as RED [53], CoDel [54], and PIE [55] drop packets when queue length or queue delay exceeds a threshold to signal congestion. Swift [56] and QCut [32] monitor the end-host queuing delay and adjust sending rate accordingly. PowerTCP [16] and HPCC [14] leverage in-network measurements at programmable switches to accurately obtain the bottleneck link state. SMUFF also works on the queue length, but it calculates its variation and decides to send how many packets so as to backlog the queue.

Wireless Transport Layer Protocol Design. Wireless links have different characteristics due to unstable channel characteristics. This motivates designs to optimize different aspects of the wireless transport. I-TCP [57] splits a TCP connection into two parts (the wireless part and the wired part) and handles frequent interruptions on the lossy wireless link. Snoop [58] performs local retransmissions over the wireless link to hide the packet loss event. TACK [18] reduces channel contention between data packets and ACKs to improve throughput. ELN [59] allows senders to distinguish between packet corruption and congestion losses and respond accordingly. SMUFF is designed to improve the network performance of the one-hop wireless link. It addresses the challenge of how to saturate a rapidly changing wireless link.

End-to-end Congestion Control Designed for Wireless Link. Wireless link speeds vary greatly over time. There are several ways to predict link speed and improve throughput. Sprout [33] observes packet arrival times to predict how many bytes can be sent from the sender. Verus [34] learns the relationship between packet delay and transmit window size and uses it to adjust the size of the transmit window. CQIC [60] and PBE-CC [61] use the physical layer bandwidth information exchanged between base stations and handsets to determine link capacity. SMUFF also needs to capture link capacity variations, but it does not need to handle the congestion on the one-hop network and can accurately track the line rate.

8 Conclusion

The rising demand for high-speed peer-to-peer mobile data transfer calls for an extremely efficient single-hop transport solution. In this work, we have addressed the challenge of achieving near wireless line rate transport based on Wi-Fi Direct by effectively orchestrating all the on-path buffers to maximize throughput while avoiding packet loss. We believe that our work identifies an overlooked yet important problem and our design can benefit other one-hop wireless networks (*e.g.,* 5G and vehicular) with different PHY/MAC protocols.

Acknowledgments

We are grateful to the anonymous NSDI reviewers for their constructive critique and valuable comments, all of which have greatly helped us improve this paper. This work is supported in part by the National Key Research and Development Plan, China (Grant No. 2023YFB2903902) and the National Natural Science Foundation of China (Grant No. 62022005, 62272010 and 62061146001). Chenren Xu is the corresponding author.

References

[1] Use AirDrop on iPhone to send items to nearby devices - Apple Support. https://support.apple.com/guide/iphone/use-airdrop-to-send-items-iphcd8b9f0af/ios.

[2] Milan Stute, David Kreitschmann, and Matthias Hollick. One Billion Apples' Secret Sauce: Recipe for the Apple Wireless Direct Link Ad hoc Protocol. In *ACM MobiCom*, 2018.

[3] Huawei share. https://consumer.huawei.com/en/support/huaweishare/.

[4] Xiaomi, OPPO and Vivo partner to bring new wireless file transfer system to global users – Mi Blog. https://c.mi.com/thread-2776586-1-0.html.

[5] What is Quick Share on Galaxy? - The Official Samsung Galaxy Site. https://www.samsung.com/global/galaxy/what-is/quick-share/.

[6] VRChat. https://hello.vrchat.com/.

[7] Abderrahmane Lakas and Moumena Shaqfa. Geocache: Sharing and Exchanging Road Traffic Information Using Peer-to-Peer Vehicular Communication. In *IEEE Vehicular Technology Conference*, 2011.

[8] Xiuxian Guan, Zekai Sun, Shengliang Deng, Xusheng Chen, Shixiong Zhao, Zongyuan Zhang, Tianyang Duan, Yuexuan Wang, Chenshu Wu, Yong Cui, et al. ROG: A High Performance and Robust Distributed Training System for Robotic IoT. In *IEEE/ACM MICRO*, 2022.

[9] Ang Li, Jingwei Sun, Pengcheng Li, Yu Pu, Hai Li, and Yiran Chen. Hermes: An efficient federated learning framework for heterogeneous mobile clients. In *ACM MobiCom*, 2021.

[10] Jinliang Yuan, Mengwei Xu, Xiao Ma, Ao Zhou, Xuanzhe Liu, and Shangguang Wang. Hierarchical Federated Learning through LAN-WAN Orchestration. arXiv 2010.11612, 2020.

[11] Daniel Camps-Mur, Andres Garcia-Saavedra, and Pablo Serrano. Device-to-device communications with Wi-Fi Direct: overview and experimentation. *IEEE wireless communications*, 2013.

[12] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, 2016.

[13] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review*, 2008.

[14] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High precision congestion control. In *ACM SIGCOMM*, 2019.

[15] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. ABC: A Simple Explicit Congestion Controller for Wireless Networks. In *USENIX NSDI*, 2020.

[16] Vamsi Addanki, Oliver Michel, and Stefan Schmid. POWERTCP: Pushing the Performance Limits of Datacenter Networks. In *USENIX NSDI*, 2022.

[17] Lynne Salameh, Astrit Zhushi, Mark Handley, Kyle Jamieson, and Brad Karp. HACK: Hierarchical ACKs for Efficient Wireless Medium Utilization. In *USENIX ATC*, 2014.

[18] Tong Li, Kai Zheng, Ke Xu, Rahul Arvind Jadhav, Tao Xiong, Keith Winstein, and Kun Tan. TACK: Improving Wireless Transport Performance by Taming Acknowledgments. In *ACM SIGCOMM*, 2020.

[19] Yossi Gilad and Amir Herzberg. Fragmentation considered vulnerable. *ACM Transactions on Information and System Security (TISSEC)*, 15(4):1–31, 2013.

[20] Wi-Fi Direct Phone Product List. https://www.wi-fi.org/product-finder-results?sort_by=certified&sort_order=desc&categories=3.

[21] IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements–Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications–Amendment 4: Enhancements for Very High Throughput for Operation in Bands below 6 GHz. 2013.

[22] IEEE Standard for Information Technology–Telecommunications and Information Exchange between Systems Local and Metropolitan Area Networks–Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 1: Enhancements for High-Efficiency WLAN. 2021.

[23] Shangqing Zhao, Zhe Qu, Zhengping Luo, Zhuo Lu, and Yao Liu. Comb Decoding towards Collision-Free WiFi. In *USENIX NSDI*, 2020.

[24] Iperf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/, 2023.

[25] iw - linux man page. https://linux.die.net/man/8/iw, 2023.

[26] Ranveer Chandra, Ratul Mahajan, Thomas Moscibroda, Ramya Raghavendra, and Paramvir Bahl. A Case for Adapting Channel Width in Wireless Networks. In *ACM SIGCOMM*, 2008.

[27] IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. 2016.

[28] Robert Braden. RFC1122: Requirements for Internet hosts-communication layers, 1989.

[29] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *ACM SIGCOMM*, 2021.

[30] RFC 793: Transmission Control Protocol, 1981.

[31] Jana Iyengar and Martin Thomson. RFC 9000: QUIC: A UDP-Based Multiplexed and Secure Transport. 2021.

[32] Yihua Guo, Feng Qian, Qi Alfred Chen, Zhuoqing Morley Mao, and Subhabrata Sen. Understanding On-device Bufferbloat for Cellular Upload. In *ACM IMC*, 2016.

[33] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *USENIX NSDI*, 2013.

[34] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive Congestion Control for Unpredictable Cellular Networks. In *ACM SIGCOMM CCR*, 2015.

[35] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. RFC 2018: TCP Selective Acknowledgment Options, 1996.

[36] Sanjib Sur, Ioannis Pefkianakis, Xinyu Zhang, and Kyu-Han Kim. WiFi-Assisted 60 GHz Wireless Networks. In *ACM MobiCom*, 2017.

[37] Shivang Aggarwal, Swetank Kumar Saha, Pranab Dash, Jiayi Meng, Arvind Thirumurugan, Dimitrios Koutsonikolas, and Y. Charlie Hu. Poster: Can Mobile Hardware Keep Up with Today's Gigabit Wireless Technologies? In *ACM MobiCom Poster*, 2019.

[38] UDP Tuning Technique from ESnet. https://fasterdata.es.net/network-tuning/udp-tuning.

[39] RFC 791: Internet Protocol, 1981.

[40] Andres Garcia-Saavedra, Pablo Serrano, Albert Banchs, and Giuseppe Bianchi. Energy consumption anatomy of 802.11 devices and its implication on modeling and design. In *ACM CoNEXT*, 2012.

[41] fillthepipe/fill-the-pipe. https://github.com/fillthepipe/fill-the-pipe.

[42] TCP optimization for network performance | Compute Engine Documentation | Google Cloud. https://cloud.google.com/compute/docs/networking/tcp-optimization-for-network-performance-in-gcp-and-hybrid.

[43] David Borman, Robert T. Braden, Van Jacobson, and Richard Scheffenegger. RFC 7323: TCP extensions for high performance, 2014.

[44] litespeedtech/lsquic: LiteSpeed QUIC and HTTP/3 Library. https://github.com/litespeedtech/lsquic.

[45] Monsoon High Voltage Power Monitor. https://www.msoon.com/online-store/High-Voltage-Power-Monitor-p90002590.

[46] Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Zhuoqing Morley Mao, Feng Qian, and Zhi-Li Zhang. A variegated look at 5G in the wild: Performance, power, and QoE implications. In *ACM SIGCOMM*, 2021.

[47] Junxian Huang, Feng Qian, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4G LTE networks. In *ACM MobiSys*, 2012.

[48] Jon Postel. RFC 792: Internet control message protocol. Technical report, 1981.

[49] Tom Herbert. BQL: Byte Queue Limits. https://lwn.net/Articles/454378/.

[50] Wi-Fi Aware | Wi-Fi Alliance. https://www.wi-fi.org/discover-wi-fi/wi-fi-aware.

[51] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *ACM SIGCOMM*, 2002.

[52] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. In *ACM SIGCOMM CCR*, 2006.

[53] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM ToN*, 1993.

[54] Kathleen Nichols and Van Jacobson. Controlling queue delay. *Communications of the ACM*, 2012.

[55] Rong Pan, Preethi Natarajan, Chiara Piglione, Mythili Suryanarayana Prabhu, Vijay Subramanian, Fred Baker, and Bill VerSteeg. PIE: A lightweight control scheme to address the bufferbloat problem. In *IEEE HPSR*, 2013.

[56] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *ACM SIGCOMM*, 2020.

[57] A. Bakre and B.R. Badrinath. I-TCP: Indirect TCP for mobile hosts. In *IEEE ICDCS*, 1995.

[58] Hari Balakrishnan, Srinivasan Seshan, and Randy H. Katz. Improving reliable transport and handoff performance in cellular wireless networks. *Wireless Networks*, 1995.

[59] Hari Balakrishnan and Randy H Katz. Explicit Loss Notification and Wireless Web Performance. In *IEEE Globecom*, 1998.

[60] Feng Lu, Hao Du, Ankur Jain, Geoffrey M. Voelker, Alex C. Snoeren, and Andreas Terzis. CQIC: Revisiting Cross-Layer Congestion Control for Cellular Networks. In *ACM HotMobile*, 2015.

[61] Yaxiong Xie, Fan Yi, and Kyle Jamieson. PBE-CC: Congestion Control via Endpoint-Centric, Physical-Layer Bandwidth Measurements. In *ACM SIGCOMM*, 2020.