Pausing and Resuming Network Flows using Programmable Buffers

Yikai Lin
University of Michigan

Ulaş C. Kozat
Huawei R&D, USA

John Kaippallimalil
Huawei R&D, USA

Mehrdad Moradi
University of Michigan

Anthony C.K. Soong
Huawei R&D, USA

Z. Morley Mao
University of Michigan

ABSTRACT

The emerging 5G networks and applications require network traffic to be buffered at different points in a wide area network with different policies based on user mobility, usage patterns, device and application types. Existing Software Defined Network (SDN) solutions fail to provide sufficient decoupling between the data-plane and control-plane to enable efficient control of where, when, and how a network flow is buffered without causing excessive control-plane traffic. Alternative approaches either cater to application-specific use cases or follow completely new paradigms that are not compatible with 5G evolution. In this paper, we present (1) a new programming abstraction for network buffering, (2) a new set of northbound APIs for network applications to pause and resume traffic flows in the network, (3) a new set of southbound APIs to scalably control where, when, and how a network flow is buffered, and (4) evaluations based on the prototype implementation of data-plane and control-plane functions. Our evaluations indicate significant performance and scalability potentials, and show near-optimal results in mobility-management and connectionless communication use cases.

CCS CONCEPTS

• **Networks → Programming interfaces**; **Programmable networks**; *Mobile networks*; *Network management*;

KEYWORDS

Software Defined Network, Network Function Virtualization, Mobility, Buffering, 5G, Cellular Network

SOSR '18, March 28–29, 2018, Los Angeles, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5664-0/18/03…$15.00
https://doi.org/10.1145/3185467.3185473

1 INTRODUCTION

As mobile networks transition toward 5G, there is an opportunity to fundamentally re-architect the core network to achieve scalability, flexibility and service agility [2, 19]. These features become especially crucial as more types of devices (IoT, Virtual Reality, Autonomous Vehicles) are joining 5G. Network Function Virtualization (NFV) and Software Defined Networking (SDN) are considered as the key enablers of these features [3]. NFV replaces hardware boxes with software instances running in VMs or containers, and allows multiple instances (of one or more NFs) to share the same commodity servers. This simplifies the development, deployment, and management of NFs, and significantly reduces the costs. SDN further simplifies the deployment and management of network services by programmatically meshing up network functions together. In SDN paradigm, logically centralized control applications programmatically change the forwarding and packet processing behavior of distributed data-plane nodes with frameworks like OpenFlow [32] and P4 [13]. These frameworks have significantly pushed the boundary of network programming. However, little advancement is made on a critical behavior of cellular networks: network buffering. Neither OpenFlow nor P4 has control over where, when and how a network flow is buffered inside the network, except for sending the flows to the controller [24].

Why is network buffering critical? Many existing services in cellular networks such as Mobility Management and Paging rely on network buffering to guarantee loss-free and order-preserving delivery during user mobility. While these services will remain fundamental in 5G networks, they will not be considered as fixed functions [46]. Instead, they will be customized for different network slices [36]. Furthermore, new set of mobile edge services that benefit from in network caching and storage should be supported when and where needed. As a brute-force approach, these services can be deployed as VNFs in central offices. However, with a proper set of abstractions for flow buffering and a set of APIs to manage flow buffering behavior, we can build a network buffering service that can be consumed easily, efficiently, and flexibly by many network services.

Towards this end, we propose Programmable Buffer (PB), a new programming abstraction for managing where, when and how a network flow is buffered within the network. PB incorporates the new programming abstractions into existing SDN programming models and provides both northbound and southbound APIs to efficiently manage flow buffering behaviors on software switches. A PB enabled switch, Programmable Buffer Switch (as will be described in Section 3.2), exposes buffering operations to the control-plane via a set of low-level (southbound) APIs. A PB service running on the SDN controller wraps up these APIs into high-level (northbound) APIs for control applications, greatly simplifying the process of buffer-based programming.

PB's low-level APIs manage the available memory on **software switches** for programmable buffers, and configure them as on-switch traffic sources and sinks. When combined with SDN's existing fine-grained flow control, applications can easily pause flows in the network, store packets for an arbitrary duration and resume/play back flows later towards any path as dictated. These functions allow PB to easily support existing services like mobility management, and enable new applications like fast mobility[1] management and connectionless communications in the 5G era.

In this paper, we make the following contributions:

- We propose PB, a novel SDN-NFV approach for managing flow buffering in a network. PB abstractions allow core network services to be further decoupled (by keeping buffering functionality on the data-plane), which helps enable a scalable high-performance 5G network.
- We design a set of low-level southbound APIs that support atomic buffer operations and are composable for high-level APIs. The low-level APIs offer precision and efficiency, while high-level APIs allow applications to easily express where and how traffic flow should be paused and resumed.
- We build a proof-of-concept prototype of PB using open-source software, and develop three applications using PB. In our benchmarks, PB shows significant performance and scalability potentials, meeting or exceeding 5G QoS standards. In simulations, PB delivers near-optimal results and show huge improvement over control-plane buffering solution. For example, PB-enabled mobility management delivers near-optimal (within 5% of the theoretical maximum throughput) results which more than double that of control-plane buffering; the decoupling between control and buffering allows PB to consistently outperform control-plane buffering in the connectionless communication use case by several orders of magnitude regardless of traffic volume.

[1]Mobility at much higher frequency, sometimes with longer handoff duration than interval, as expected in 5G networks. Will be described in detail in Section 5.1.

2 WHY PROGRAMMABLE BUFFER?

We first draw a clear distinction between Programmable Buffers and legacy switch buffers (queues). **Switch buffers (queues)** are part of the switch processing pipeline. They *temporarily* hold packets while packet schedulers decide in what order and when to serve these packets. They absorb arrival rate fluctuations to prevent packet loss, or enforce QoS metrics. **Programmable Buffer** is a storing unit alongside the switch processing pipeline. PBs serve as on-switch traffic sinks (in the case of "pausing") or sources (in the case of "resuming") when attached to the pipeline and they can hold the packets indefinitely unless otherwise instructed by the control-plane.

In many scenarios, packets are required to be buffered until a certain event happens (e.g. mobile device reconnects to a new base station). Such events are typically not switch-local and require a global view to manage multiple buffers in different parts of the network in a coordinated manner.

Moving toward 5G and embracing different types of mobile devices and applications, we will see an increasing demand for buffering support. As SDN and NFV become the building blocks and key enablers for next-generation networks, we can leverage them to make network buffering programmable and provided as a general function for all applications just like forwarding and packet processing. This also echos with the vision of network slicing [36] which partitions network architectures into virtual elements with different requirements. Here we list three motivating examples to show that the requirement for network buffering varies depending on the application and device types. Therefore, catering for application-specific use cases will not fit all and will end up reinventing wheels.

2.1 Network Buffering Examples

Mobility Management. In cellular networks, handling user mobility is a core service. Since packet loss and out-of-order delivery have a severe performance impact on TCP connections, the network needs to buffer (pause) users' downlink traffic before they disconnect from the base station, and resume it after they reconnect to another base station. End host applications are agnostic of such mobility or buffering behaviors, which are managed by the network control plane.

Network Function Flow Migration. With NFV, scaling a service simply requires the network to instantiate new NF instances. However, as pointed out in [24], many stateful network functions (e.g. Bro IDS) require current states of the old NF instance to be transferred to the new instance, which must be completed prior to flow migration for correctness.

This requires the network to actively buffer the live traffic while state transfer is happening.

Connectionless Communication. As 5G approaches, new communication paradigms such as device-to-device communications need to be supported efficiently. Such communications should work even when one end is offline or in mobility. Connectionless communication accommodates such special requirements by allowing the network to serve as a surrogate receiver, freeing the sender from buffering the data. This is even more crucial when the sender is battery constrained (e.g. a sensor) and the receiver (e.g. data analysis server) only goes online periodically. In this case, buffering requests originate from end hosts instead of the network.

2.2 Related Work

Switch buffer management is a well-studied topic dating back to the 70s [27]. Most previous works focused on designing packet scheduling algorithms or switch buffer architectures for specific use cases. More recently, to address the lack of programmability of packet scheduling, Sivaraman et al. [42] proposed a push-in first-out (PIFO) queue as a basic abstraction for programmable scheduler; Kogan et al. [31] proposed a framework for constructing custom switch buffer architecture; HotCocoa [20] implemented entire congestion control algorithms in programmable NICs. As explained above, Programmable Buffer and switch queue have different purposes. Switch queues are not designed to hold packets indefinitely and packet schedulers have no visibility beyond switch-local events. In addition, these solutions require new scheduler/architecture/algorithm to be recompiled which changes the switch processing pipeline.

On-switch storage. NetCache [29] proposed to use hardware programmable switches as key-value caches for load balancing storage server requests. In particular, the authors utilize the register arrays implemented with on-chip memory to store *small* values. Such hardware switches have limited/fixed number of registers and register size (128B) that are not well suited for PB which dynamically manages the number and sizes of buffers to store packets. In this work, reading and writing the cache are triggered by certain packets arriving at the switch, whereas PB allows the controller to programmatically turn a buffer into traffic source or sink.

Network storage. Plank et al. [38] presented the Internet Backplane Protocol for network applications to actively utilize storage as part of the network fabric. This work differs from PB in many ways. Firstly, it is a distributed approach while PB exercises SDN's centralized model. Secondly, it treats network storage as a passive function triggered only by end-hosts, which cannot address scenarios where buffering necessity originates from within network (e.g. user mobility). Thirdly, end hosts lack the global view to optimally decide where to store. However, this paper does share the same vision with PB of the importance of in-network buffering for many applications.

Packet buffering in SDN. Without data-plane buffering support, SDN applications have to rely on the controller to temporarily store the packets [24], thus placing it on the critical path of network flows. This could incur high per packet latency [23], and occupy limited controller resources (memory, CPU and control channel bandwidth) [44], which suffers on scalability. PB addresses this problem by decoupling the control from buffering and keeping latency-sensitive operations on the data-plane: the controller is not on the critical path of network flows and is only involved to initiate "pause" or "resume" instructions (which, as shown in Section 3.4, are often accomplished with only one control message).

Cellular SDN/NFV network architectures. Recently, there has a been substantial focus from academia and industry on realizing the network architecture of 5G cellular core networks based on the SDN/NFV concept [8, 21, 34, 39, 41]. In particular, SoftCell [28] departs from the centralized policy enforcement in the core network by directing users' traffic through distributed middleboxes. SoftMoW [33, 35] builds a scalable control plane using the hierarchy technique to enable global network optimization. These SDN/NFV architectures are complementary to PB in terms of scaling the control-plane and handling failures.

3 PB FRAMEWORK

3.1 Overview

PB framework extends from the typical SDN-NFV paradigm with a Buffer Engine on each data-plane node in coordination with a Buffer Service on the control-plane (See Figure 1). In addition to the default southbound APIs (interface 4 in Figure 1) such as OpenFlow [32] and P4 [13], Buffer Service communicates with Buffer Engine via PB's southbound APIs (interface 3). These low-level APIs are wrapped up by Buffer Service as higher level APIs for upper layer control applications (interface 1). Together with the built-in flow management capabilities (interface 2), a control application can create buffers when and where desired and direct traffic flows into/out of buffers.

Inline with the 5G vision, elements of the PB framework can be mapped to different functions in the 5G architecture: programmable buffer as User Plane Function (UPF), and Buffer Service as Session Management Function [16].

Next, we will break down the PB framework and describe the design of each in details. We start from the data-plane switch abstraction, Programmable Buffer Switch (Sec. 3.2). On the switch are the two programming abstractions, programmable buffer and virtual port. Then we introduce the southbound buffer APIs (Sec. 3.3) that are used to orchestrate and monitor the states of programmable buffer and virtual

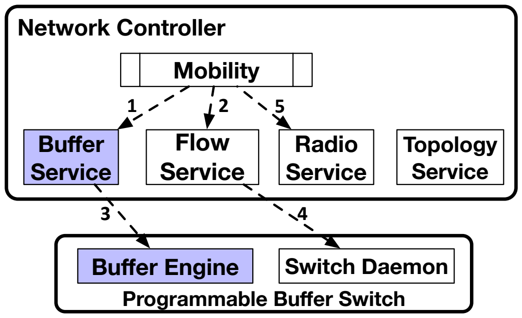

Figure 1: PB Framework Architecture

port. Buffers in different states (Sec. 3.4) carry out different functionalities. Control applications manage these states and fulfill their intentions through the northbound APIs (Sec. 3.5).

3.2 Programmable Buffer Switch (PBS)

In its essence, Programmable Buffer Switch (PBS) is a buffer-enabled SDN switch. Besides the typical SDN switch composition (a pipeline of match+action tables, an on-switch agent/daemon, and external ports), PBS is comprised of programmable buffers (buffers) and virtual ports (vports). To manage the buffers and vports, a PBS implementation can choose to either have a PBS agent (buffer engine) alongside the default switch agent (See Figure 2) or extend the switch agent with buffer/vport managing capabilities. As per the SDN paradigm, buffers, vports, and match-action table entries are dynamically created, configured and removed by control applications running on top of the network controller. Naturally, PBS can fall back to a regular SDN switch and be completely backward compatible with SDN applications that do not utilize PB APIs.

Programmable Buffers: Programmable buffers serve as data-plane storage buckets on PBS nodes. Distinct buffers have memory isolation and each buffer has an initial memory size specified at its time of creation and re-configurable later by control applications. To make them really instrumental, the controller must create vports that bind buffers to the switch (See buffer-1 and vport-1 in Figure 2). By default, buffers store packets in the order they are received and simply implement a FIFO queue. They can also be configured with other queuing polices (e.g. priority queue). Buffers do not perform any manipulation of packet contents and any such manipulation is done by the switch processing pipeline (OpenFlow or P4) before the packets enter a buffer or after they exit a buffer.

Virtual Ports: Vports are software interfaces in PBS that bind buffers to a PBS node. Vports share the same 'port' abstraction with external ports, i.e., its one end is attached to the switch processing pipeline and the other end is either attached to another entity (in this case a buffer) or free. Although a vport is bidirectional, we intentionally give each

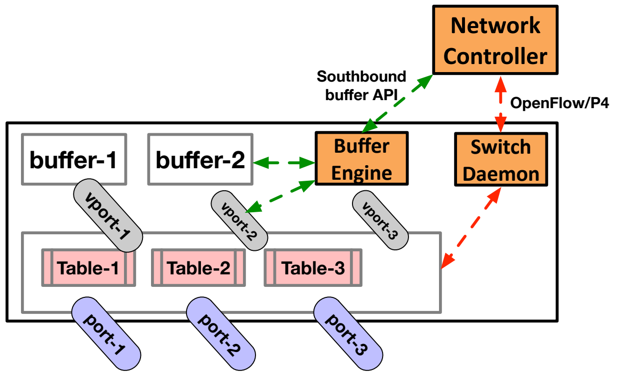

Figure 2: PBS Architecture

vport two modes of operations: *RX, TX*[2]. In *RX* mode, packets coming from the switching pipeline (following a flow table entry) will enter the buffer that is bound by the vport. In *TX* mode, packets in the buffer are sent to the switching pipeline. One could think of buffers as virtual hosts from the switch's point of view. Thus, it is possible to bind multiple vports to a buffer. Typically, however, one (in RX mode) or two (one in RX and the other in TX mode) vports are attached to a given buffer.

Vports and buffers themselves do not have the notion of what a "flow" is. Vports fill and empty buffers, while buffers queue packets without interpreting their headers or payloads. Therefore, by specifying a match-action table entry, the controller determines which set of packets should be sent to or retrieved from particular buffers. In each table entry, vports are specified as in-ports or out-ports depending on the flow direction.

Note that once a buffer is created and bound to switch by vports, the controller can reuse the same buffer for any other network flow by simply modifying the match-action table entries. For instance, a buffer initially used to store all packets destined for mobile subscriber Alice can later be used to store all packets destined for mobile subscriber Bob. If desired, the same buffer can be used to store both Alice's and Bob's network traffic at the same time.

3.3 Southbound Buffer API

For southbound communications between the Buffer Service and Buffer Engines, we intend to make the APIs stable and atomic (composable), since they support the most fine-grained buffer operations.

As shown in Figure 2, Buffer Engine exposes programmability and monitoring capabilities to authorized external controllers through the PB southbound APIs. Table 1 shows the

[2]Explicit mode configuration is actually quite useful: it allows the vports to have certain access control for traffic going through buffers. For example, when a vport is in *TX* mode, any packets coming from the pipeline to this vport will be dropped.

APIs to orchestrate the state of a buffer and vport, and to query/subscribe buffer/vport status. The function of each API call is pretty self-explanatory. One important issue to point out is that the number of control messages transmitted through the southbound PB channel does not necessarily equal that of API calls, which could be quite large due to their fine granularity. By bundling several API calls in one control message, the overhead (delay) of actually performing that many API calls could be further reduced. For example, if a control application instructs the Buffer Service to create three buffers *B1,B2,B3* at switch *S1*, it generates only one control message instead of three. We assume no bundling throughout the rest of the paper and leave it for further study.

Next, we show the five typical states of a buffer and how each transitions into another. These state transitions are managed by the southbound buffer APIs.

3.4 Programmable Buffer States

There are five typical states of a buffer. Figure 3 depicts the simplest scenarios of each state.

Buffering State: A buffer is bound to a switch by *only* RX mode vports, which means there is only inbound traffic towards the buffer. As mentioned in the last section, in order for traffic to be steered into the buffer, there should be match-action table entries specifying these vports as outport. Inside the buffer, packets will be placed at the end of the FIFO queue unless otherwise configured by the controller. If the buffer exceeds its capacity, either new packets will be dropped (from the tail) or oldest packets will be dropped (from the head) based on the buffer configuration set by its control application.

Serving State: Opposite to the Buffering state, *only* TX mode vports are binding the buffer to the switch. Packets will be removed from the head of the FIFO queue, sent out via the attached vports, and processed through matching table entries.

Forwarding State: When *both* RX mode and TX mode vports are bound to the buffer, it is in the Forwarding state. Incoming packets will be placed at the end of the FIFO queue while the oldest packets will be removed from the head.

Free State: If an *empty* buffer has *no* vports bound, it is in Free state. A buffer starts in Free state when first created; a buffer in Serving state transitions to Free state if it is completely emptied and unbound from vports. A Free state buffer is essentially a resource that can be recycled.

Storing State: If a *non-empty* buffer has *no* vports bound, it is in Storing state instead. A buffer in Buffering state transitions to Storing state if it is unbound from its vports. In this state, buffers hold the packets as long as the switch exists and the buffer is not removed by the control application.

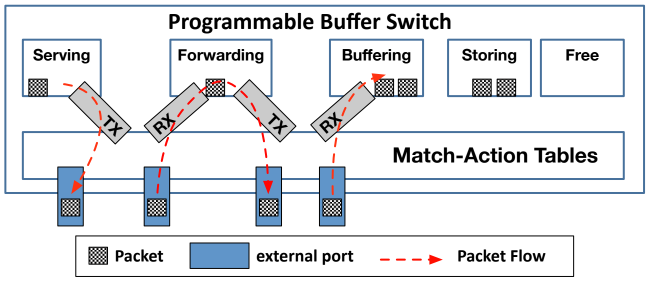

Figure 3: Five different states of a buffer.

Transitions between different states occur as a result of one or more API calls listed in Table 1. A buffer starts in the Free state with *create_buffer* command. It can transition to Buffering state or Serving state with *create_vport* (if needed) and *bind_buffer_to_vport*. A Buffering state buffer can transition to Forwarding state if a TX mode vport is bound, or to Storing state if all vports are unbound with *unbind_buffer_from_vport*, etc.

Note that, because of the explicit vport mode design, a **Forwarding State** buffer can transition to **Buffering State** by simply changing the *TX* vport to *RX* mode, and vice versa, without even modifying existing match-action table entries. As later shown in the buffer-enabled applications, this allows the controller to be minimally involved (sending one control message) and more scalable.

3.5 Northbound Buffer API

Common SDN controller implementations [5, 10, 12, 14] come with basic network services like topology discovery and flow management. These services provide northbound APIs to upper layer control applications for managing data-plane nodes while saving them from the troubles like discovering the topology or crafting a control message from scratch. Following the same paradigm, Buffer Service provides high-level buffer APIs to control applications to decide where, when and how a network flow is buffered.

These APIs can be divided into two levels: task-level and intent-level. **Task-level APIs** are directly composed by southbound buffer APIs that execute in a certain order to carry out a common task. In Code Sample 1, we present two task-level APIs that carry out two most common tasks in buffer-enabled applications: *create a buffer in a given state* and *change a buffer to a given state*. They are both solely composed by southbound buffer APIs introduced in Section 3.3. Buffer Service maintains a copy of the states of buffers and vports, thus in the second function *bufferStateChange* an application does not need to provide the current state of the buffer that it's operating on.

On top of the task-level APIs, we present two **intent-level APIs**: *pause_flow()* and *resume_flow()*. Both APIs composed of not only task-level buffer APIs, but also flow APIs.

Table 1: Southbound Buffer APIs

API	Parameters	Notes
create_buffer()	device_id, size, queue_type	queue_type: default to FIFO
create_vport()	device_id, mode, [port_num]	vport mode: {RX, TX}
bind_buffer_to_vport()	device_id, buffer_id, vport_id	Bind the given buffer to given vport
unbind_buffer_from_vport()	device_id, buffer_id, vport_id	Unbind the given buffer from the given vport
set_vport_mode()	device_id, vport_id, mode	Change the mode of the given vport
remove_buffer()	device_id, buffer_id	Remove the given buffer
remove_vport()	device_id, vport_id	Remove the given vport
query_buffer()	device_id, object, [rule]	object: what to query, e.g. buffer utilization.
query_vport()	device_id, object, [rule]	rule: notify subscribed application when met

```
// Task-level Northbound Buffer APIs
import pbService as pb

def createBuffer(swID, state, [buffer params...])
  // create a new buffer
  buf = pb.create_buffer(swID, [buffer params...])
  // create vports and bind buffer to them
  switch state:
    case Buffering:
        vp = pb.create_vport(swID, RX)
        pb.bind_buffer_to_vport(swID, buf.id, vp.id)
    case Forwarding:
        vp1 = pb.create_vport(swID, RX)
        vp2 = pb.create_vport(swID, TX)
        pb.bind_buffer_to_vport(swID, buf.id, vp1.id)
        pb.bind_buffer_to_vport(swID, buf.id, vp2.id)
    // other cases...

def bufferStateChange(swID, bufID, state)
  buf = pb.getBuffer(bufID)
  switch buf.state and state:
    case Free and Buffering:
        vp = pb.create_vport(swID, RX)
        pb.bind_buffer_to_vport(swID, bufID, vp.id)
    case Buffering and Forwarding:
        vp = pb.create_vport(swID, TX)
        pb.bind_buffer_to_vport(swID, bufID, vp.id)
    case Buffering and Serving:
        pb.set_vport_mode(buf.vports[0], TX)
    case Forwarding and Buffering:
        pb.set_vport_mode(buf.vports[1], RX)
    case Buffering and Storing:
        pb.unbind_vport_from_buffer(swID, bufID, buf.
    vports[0])
    // other cases...
```

Code Sample 1: Task-level Buffer APIs

pause_flow(sw_id, in_port, flow_filter, [buffer_id]). Through this API call, control applications decide where (i.e. which Programmable Buffer Switch) to buffer what flow coming from which port. Buffer_id is optional. If it is not provided, Buffer Service will automatically allocate an unoccupied buffer or create a new buffer. Depending on the status of the target flow, Buffer Service might add a new match-table entry to redirect the flow into the buffer, or, if the flow is going through a buffer already, simply change the state of that buffer to Buffering. And as shown in Section 3.4, changing a buffer from Forwarding state to Buffering state could be as simple as just setting the TX vport to RX mode.

resume_flow(sw_id, out_port, flow_filter, [buffer_id]). This API call allows control applications to turn Storing state buffers into traffic sources or resume flows in a Buffering state buffer by changing it to Forwarding mode. If a buffer_id is not provided, Buffer Service will try to locate the buffer used for storing the flow and redirect its content to the out_port. Otherwise, it will add one or more table entries for packets coming out of the given buffer. Note that flow classification happens twice in the second case, since flow filters used for *pause_flow()* do not necessarily need to match those used for *resume_flow()*. Applications could simply allocate a huge buffer to store flows coming from different sources, and later decide which sub-flows go to which destination.

4 SUPPORTING EXISTING APPS

Network buffering is a critical function required by many existing applications. This section introduces two of these applications in more details and show how they can be supported in a PB enabled network.

4.1 LTE Mobility Management (LMM)

As described in Section 2.1, LTE mobility management is complex and requires flow buffering and routing across multiple nodes. The initial phase involves radio signal measurements and reporting by the UE to its current base station (Source eNB). Source eNB makes handoff decision based on these measurements and requests handoff from a Target eNB. If request is admitted, the Source eNB starts buffering the downlink packets and instructs UE to establish a radio connection with the Target eNB. The Source eNB sends its buffered and in-transit packets coming from the Serving Gateway (S-GW) to the Target eNB. The Target eNB buffers these packets until radio connection is set up for the UE. In parallel, the Target eNB through the Mobility Management Entity (MME) performs a path switch from the S-GW to itself for all future downlink traffic. To preserve packet order, Target eNB buffers all the packets from the new path until all the buffered and in-transit packets from the source eNB are served. To facilitate the detection of last in-transit packet, S-GW transmits a special packet with End Marker. Once the marked packet is received by the Target eNB, it

```
// LTE Mobility Management with PB
import pbService as pb
import flowService as flow
import radioService as radio

self.on(HandoffStart, function(event)
    // buffer 0 at source eNB
    buf0 = event.sourceBS.buffers[0]
    // buffer 1,2 at target eNB
    buf1 = event.targetBS.buffers[1]
    buf2 = event.targetBS.buffers[2]
    // direct traffic from old path to target eNB
    flow.FlowMod(buf0.vports[1], buf1.vports[0])
    // detach UE
    radio.detach(event.ue, event.sourceBS)
    // switch path at anchor switch
    flow.FlowMod(anchorSW.port[2], buf2.vports[0])

self.on(HandoffEnd, function(event)
    // ue attaches to target eNB
    radio.attach(event.ue, event.targetBS)
    // turn buffer 1 into Forwarding State
    pb.set_vport_mode(buf1.vports[1], TX)

self.on(IndicatorReceived, function(event)
    // turn buffer 2 into Forwarding State
    pb.set_vport_mode(buf2.vports[1], TX)
```

Code Sample 2: PB-enabled LMM Application

starts serving the buffered and in-transit packets from the new path. This whole process is in place to ensure loss-free, order-preserving packet delivery for good TCP performance.

In PB-enabled networks, the PBS data-plane abstraction applies to eNBs as well. LTE Mobility management (LMM) application runs on the controller and channel measurements from the UEs as well as load information from the eNBs are passed onto this application. LMM makes handoff decision, target eNB determination, and admission control based on these information. LMM sets up one programmable buffer at Source eNB, where the UE is currently attached to, and two buffers at the Target eNB, one for packets coming from Source eNB and the other for downlink traffic coming from the new path (Figure 4). All buffers are initially created in the *Buffering* state. Once buffers are set up, LMM instructs the UE to detach from the current eNB and attach to the new eNB. It instructs the anchor switch to switch from old path to new path while instructing Source eNB to change the state of its buffer to *Forwarding*. At this point, all the buffered and in-flight packets coming to Source eNB are diverted toward the first buffer at Target eNB (green arrow in Figure 4). The second buffer at Target eNB is still in the *Buffering* state and hence all new path packets are buffered in this second buffer. Once the radio link is established between UE and Target eNB, LMM receives the completion signal upon which it instructs Target eNB to change the first buffer state to *Forwarding*. Hence, two buffers at Source and Target eNBs are in tandem serving UE the in-transit traffic coming via old path. To ensure old path is cleared, when switching the path, LMM also injects an indicator packet at anchor switch (for example, using a packet-out message in OpenFlow protocol). The indicator packet traverses anchor switch and Source eNB before

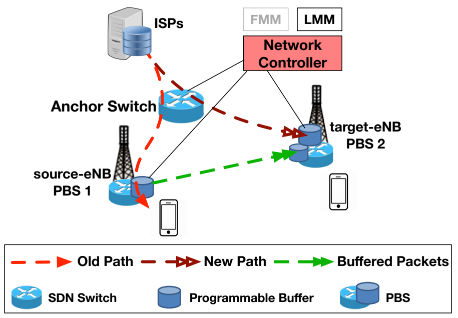

Figure 4: PB-enabled LTE Mobility Management

reaching Target eNB as the last in-transit packet from path A. Since this packet is marked and every eNB is programmed to notify the controller upon receiving it, LMM knows that there are no in-transit packets left from path A. Now, LMM instructs Target eNB to change the state of the second buffer to *Forwarding* state. UE starts receiving packets from path B. Code Sample 2 shows part of LMM in pseudo code.

4.2 NFV Flow Migration

NFV flow migration can be considered as a special version of mobility management, as the mobility happens inside the network instead of network edge. This use case is specifically addressed in [24]. The authors first started using the SDN controller to buffer in-flight packets during NF state transfer, which leads to triangular routing and many other performance and scalability issues. They later adopted an alternative approach [23], which instead requires the NF instance to buffer in-flight packets before state transfer finishes.

With PB's support, the controller only needs to set up a Buffering State buffer for each new NF instance at the same switch, and chain the buffer with its corresponding NF instance with flow rules. This way, the buffer can be independently managed by the controller and adjusted according to different traffic volume and pattern without having to modify the NF programs. Packets will stay inside the buffer and be immediately available when state transfer finishes.

5 ENABLING NEW APPS

With PB's APIs and abstractions, we can achieve more than supporting existing applications. In this section, we introduce two new network applications that PB enables: Fast Mobility Management and Connectionless Communications.

5.1 Fast Mobility Management (FMM)

With 5G envisioning massive bandwidth improvement over 4G, the current radio access link technology in LTE networks is no longer viable. This has researchers look into alternatives such as Millimeter-Wave (mm-wave) and much denser cellular deployments [40]. Abundant spectrum of high frequencies and densification resolve the bandwidth shortage

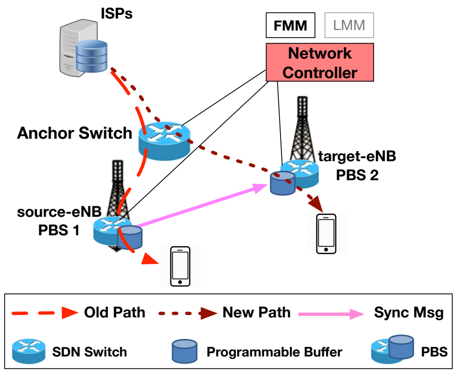

Figure 5: PB-enabled Fast Mobility Management

problem, but such systems also require high directionality and narrow beam widths, imposing major challenges in mobile scenarios. Consider the scenarios where many roadside or lamp-post base-stations are deployed with 10 to 20 degree beam-widths. The beam-training takes 10s of milliseconds [43], and even at moderate vehicular speeds (e.g., 30 mph) with 5-meter separation between the base station and road lanes, the residence time in each base station becomes comparable to the beam-training time.

This becomes problematic for the LMM application. In that, after each handoff, the second buffer at the target eNB (See Figure 4) has to wait till the first buffer is emptied (old path is clear) before it can start serving the UE. In theory, the time it takes for the first buffer to empty is equal to or larger than the handoff duration, because the first buffer stores all in-flight packets during the handoff period. In practice, due to control latencies, this time is even longer. Simply put, LMM will not work under such extreme conditions because residence time with a eNB could be shorter than handoff duration. Therefore, we propose a new mobility management solution that eliminates inter-eNB traffic forwarding and supports much higher handoff frequencies, called Fast Mobility Management (FMM).

This application takes a more aggressive approach to ensure packets are always ready whenever and wherever a UE attaches, and it can detach anytime it wants. To enable this, when a UE is attached to a particular eNB, all neighboring eNBs (or a subset of them based on predictions of mobility pattern) will be receiving downlink packets from the anchor switch and buffering them. In other words, the anchor switch is multicasting UE packets to all potential target eNBs. When the UE moves and reattaches to another eNB, the buffer there can immediately start serving the UE without having to wait for the source eNB to forward the buffered packets. Without buffer programmability, this kind of dynamic service cannot be orchestrated unless the software in each base station is upgraded. With PB, however, this application can be easily supported as follows.

The FMM application provisions a buffer in *Forwarding* state at the UE's current attachment point (i.e., Source eNB-PBS). At the same time, FMM provisions buffer for the same UE in *Buffering* state for each potential next base station (i.e., Target eNB-PBS nodes). FMM also installs a forwarding rule at the anchor PBS node of all these base stations to multicast the UE traffic to Source and Target eNB-PBS nodes before any handoff decision is taken. Thus, packets are buffered at Target eNB-PBS nodes. Before UE starts detachment, FMM changes the state of UE's buffer at Source eNB-PBS to *Buffering*. After the reattachment, FMM changes the state of UE's buffer at the new eNB-PBS to *Forwarding* and UE can start receiving packets from it. After the handoff, FMM can update the set of Target eNB-PBS nodes, thus accordingly change the multicast group at anchor PBS while terminating/recycling the buffers provisioned for UE at eNB-PBS nodes that are no longer potential targets.

Inter-buffer Synchronization: Since in FMM each buffer keeps their own copy of the packets, packet-loss or duplicates become an issue. For example, if the target-eNB buffer is sufficiently large, we should expect the first packet in it to be older than the head-of-line packet of the TCP session. In other words, there will be duplicate packets in the target buffer. In contrast, if the target buffer is too small, there will be packet loss. Both duplicates and losses can lead to inferior TCP performance. To resolve this problem, there needs to be a synchronization mechanism to align the target buffer head with the source buffer head. That is, when handoff happens, as the source buffer stops sending, the target buffer should know what the last sent packet was and purge any packets older than it. Obviously, this only works when the target buffer is sufficiently large (has duplicates), since there is no way to recover lost packets. Such synchronization only needs to convey a unique packet identifier from the source buffer to the target buffer, which is negligible compared to LMM's traffic redirection. In our experiments, we find 2 bytes of IPv4 id and 2 bytes of transport layer checksum to be reliable for uniquely identifying packets even when encrypted. Since control applications can decide which buffer implementation to use for their traffic, they can choose the right identifiers based on the traffic pattern. Figure 5 depicts a handoff scenario with FMM and synchronization enabled.

FMM vs. LMM: In the 5G era, as network slicing [36] becomes the norm, different mobility management solutions like FMM and LMM are expected to run on the same infrastructure serving different devices and users based on their needs. Compared with LMM, FMM allocates more buffers for each user since it uses multicast at the anchor switch to ensure immdediate packet availability, which has higher buffering overhead. Therefore, FMM targets a small portion of users that are travelling at a high speed and thus handoff much more frequently. In terms of control overhead, FMM generates the same amount of control messages during each handoff as LMM (managing the multicast group is outside the control loop of handling handoffs).

5.2 Connectionless Communications

Connectionless network services which are, e.g., used to support Internet of Things (IoT), have been one of the key use cases discussed for next generation mobile networks [30]. PB abstractions can be utilized by connectionless services to have a slice of the underlying transport fabric as a caching and content distribution infrastructure and enable asynchronous communications between devices (device-to-device communications).

To store/upload any content, the control application first associates the content with a network flow. How this association is done using which packet header fields is implementation specific. Once the one-to-one association between contents and network flows is done, to store a particular content on a given PBS node, control application should create a distinct buffer for the content (i.e., for the corresponding network flow) at the given PBS node in *Buffering* state. After it ensures that all the flow packets are stored, the control application can transition the buffer into *Storing* state (Figure 3).

If stored content is requested by another node in the network, the IoT or content distribution application first sets up a routing path and simply transitions the buffer of that content into *Serving* state, which will automatically start serving the requesting node.

6 EVALUATION

In this section, we present our prototype of Programmable Buffer using open-source software and its scalability and performance gains compared with alternative SDN solutions.

6.1 Prototype and Methodology

We prototype **Buffer Service** as a Ryu [14] controller module which provides the northbound buffer APIs to other applications. We implement **Buffer Engine** in C++ as a process running alongside the software switch. Buffer Service establishes connection with Buffer Engine via gRPC [6]. The engine serves as both gRPC client and Docker agent. **Programmable buffers** are packaged as Docker [4] containers managed by Buffer Engine via Docker APIs. Each container runs a C program that receives IPC calls (e.g. bind/unbind) from Buffer Engine. The motivations behind containerizing buffers, besides ease of managing resources like CPU and memory, is that control applications can choose between different buffer implementations by specifying a Docker image. This allows different queue types and application-specific tweaks (e.g. inter-buffer synchronization in Section 5.1) to be pre-built and used flexibly.

We use two open-source **Software Switch** packages, OpenvSwitch [11] and mSwitch [26], and two companion **Virtual Port** implementations, TAP [17] interfaces and mSwitch ports respectively. OpenvSwitch is OpenFlow-compatible and thus used in our simulations (Section 6.3); mSwitch is picked for its high port density and used in scalability tests (Section 6.2.2). We also implement two packet processing techniques for programmable buffers: Memcpy and Zerocopy. Memcpy, as its name suggests, uses the Memcpy() function to copy packets from/to vport packet queues. Data copy incurs high overhead under high bit rates. We thus implement a zero-copy programmable buffer with netmap [9] that allows us to preserve packet buffers out of packet I/O queue without data copy. We use this variant of programmable buffer on top of mSwitch, and demonstrate superior performance in Section 6.2.2. We also modify mSwitch to not perform data copy between its ports to further reduce the overhead. Programmable buffer size is set to 1024 packets for benchmarks and 10000 for simulations. Unless otherwise specified, we use an Ubuntu 16.04 (Linux 4.13.0) Desktop with Intel core i7-7700K@4.2GHz quad-core processor and 16GB of RAM for our experiments.

6.2 Benchmark Results

It is widely acknowledged that future 5G networks should be able to support high-bandwidth applications like 4K video streaming, Virtual Reality and Augmented Reality while incurring ultra-low latency ([1, 2, 36]). Many believe the user experienced data rate should be at least 50Mbps and up to 1Gbps depending on coverage and resource availability. According to [18], for 5G radio access network, the control-plane latency should be less than 10ms and data-plane latency should be around 1ms. The decoupled design of PB allows it to scale up independently on the control-plane and data-plane depending on the workload. To see if each component of PB can meet these QoS metrics, we run several benchmarks as described below.

6.2.1 Packet Rate and Latency. Since programmable buffers are on the critical path of user traffic, we perform the following experiments to get the steady packet rate and one-way latency of single programmable buffer with varying packet sizes. On the machine, we create a *Forwarding* state buffer, two pkt-gen applications (from netmap) as traffic source and sink, and connect them with two virtual ports. We set the two pkt-gens to transmit and receive mode respectively, and specify different packet sizes in each run. The three processes are pinned to three CPU cores, and the batch size is set to 512. We configure the virtual ports to operate as netmap pipes[3] and there are no packet losses. The results are shown in Figure 6. As expected, buffer with zerocopy has constant high

[3]A shared memory packet transport channel supported by netmap.

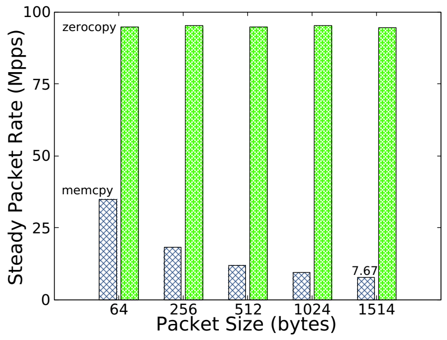

Figure 6: Buffer Packet Rate with Varying Packet Size

Table 2: PB Latency Measurements

One-way Delay (us)	Memcpy	5.28-6.00
	Zerocopy	5.43-5.82
API Call Execution Time (ms)	create_buffer	39.9
	create_vport	11.7
	bind_buffer_to_vport	0.4
	set_vport_mode	1.5
	unbind_buffer_from_vport	1.4
	flow_mod	1.3
	remove_vport	27.0
	remove_buffer	39.7

packet rate (90+ Mpps), while memcpy incurs higher overhead as packet size increases, though still achieving 90+ Gbps throughput with 1514-byte packets. This indicates that both programmable buffer implementations are fairly efficient.

Similarly, we measure the one-way packet delay of programmable buffers by setting the pkt-gens ping and pong mode respectively (packets travel a round trip and the RTT is calculated at the ping side). We also use various packets sizes up to 1514 bytes. The results are shown in Table 2. For both zerocopy and memcpy versions with packet size up to 1514 bytes, programmable buffers incur no more than 6 microseconds one-way packet delay, which is more than two orders of magnitude smaller than the 1-ms 5G standard [18].

6.2.2 Data-plane Scalability. As 5G aims to serve massive number of devices, scalability becomes one of the most critical criteria when evaluating the design of PB. We start our scalability analysis by measuring the resource footprint of programmable buffers which correlates with the number of users one PBS can serve. In this test, we use the same setup as in the last benchmark (with buffer on path, 1514-byte packets), but instead of saturating the bandwidth, we vary the throughput and measure the corresponding CPU usage. Our results show that at 40Gbps throughput, the memcpy version of buffer consumes 48.2% CPU while zerocopy version consumes only 5%. Both versions consume 0% CPU when idle, which is expected.

To better understand how well PBS scales to larger number of users (flows), we carry out another experiment to simulate up to 2048 active users (flows) simultaneously. Due to inherent CPU core requirements to host a large number of programmable buffers, we employed another Linux (4.11) server with Intel Xeon E5-2690v4@2.6GHz 14-core processor and 64GB of RAM (4 cores assigned to traffic source, 4 to buffers and 1 to traffic sink). Each flow is handled by a different buffer, totaling up to 2048 programmable buffers on one PBS instance. We write a simple routing module for mSwitch that (1) distributes incoming packets (from traffic source) to each buffer based on destination IP address, and (2) aggregates packets to one port (traffic sink). We use mSwitch with this module enabled to connect traffic source and sink with up to 2048 buffers, and record the aggregate throughput as shown in Table 3. Although the absolute values could vary on different machines, these results clearly demonstrate the data-plane scalability of PB.

Table 3: Concurrent Flow(Buffer)s on PBS

# of Flows	4	16	64	256	1024	2048
Throughput (Gbps)	187	123	121	110	105	102

Finding 1. PBS can deliver 100+Gbps throughput with over 2000 flows using 4 cores on a commodity server, attaining 50 Mbps per flow that is twice the recommended bandwidth for 4K video streaming [7].

Another factor contributing to programmable buffer's resource footprint is memory. PBs have marginal memory overhead on top of what is needed for storing the packets. As such the memory requirement becomes simply the product of forwarding speed and the duration of traffic interruption. E.g., for 100 Gbps bandwidth, it will be several to tens of GBs. We argue that the amount of RAM available on a commodity server (e.g. 128GB) is more than enough to satisfy the memory requirements for traffic interruptions at the scale of tens [25] to hundreds of milliseconds [24]. Further, PB dynamically (re)allocates buffer memory, unlike existing solutions with pre-determined buffer size based on estimation [45].

6.2.3 Control-plane Scalability. In the second row of Table 2, we show the RTT measurements[4] of each PB API call and FlowMod [15] of OpenFlow. Our test application repeatedly calls each southbound API and measures the time to complete each call. In our measurements, operations related to create/remove buffers/vports have higher overhead than simply binding vports to buffers and modifying vport modes. This is expected since the former operations involve memory (de)allocation. As mentioned earlier, buffers and vports have minimal idle resource footprints, thus provisioning them beforehand can effectively remove their presence on the critical path of latency-sensitive operations. Compared to FlowMod,

[4]We colocate the controller and PBS to minimize communication latency.

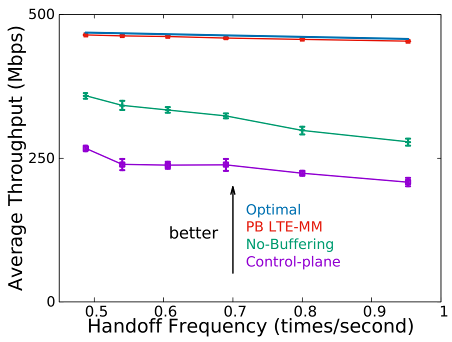

Figure 7: Average Throughput of different LTE mobility management solutions on low handoff frequencies

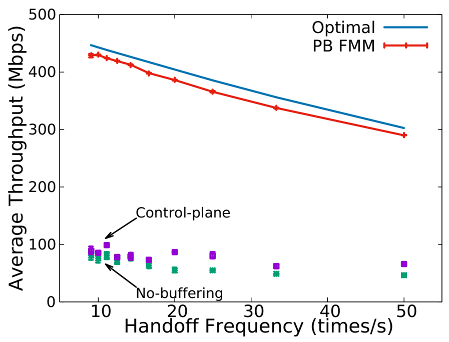

Figure 8: Average Throughput of fast mobility management solutions on high handoff frequencies

other buffer API calls have similar or smaller latencies. As discussed in Section 2.2, many previous works have focused on improving the scalability of SDN control plane to support carrier-grade workloads [10, 28, 35], which are complementary to PB's design. For existing SDN applications, PB is fully backward compatible and thus shares the same level of scalability; for buffer-based applications (such as LMM shown in Code Sample 2), since buffer API calls have similar costs, and the number of these calls is minimized[5], the control-plane scalability is not much impaired.

6.3 Simulation Results

6.3.1 LTE Mobility Management. As described in Section 4.1, PB allows critical core services to be virtualized while keeping low-latency data-plane buffering functionalities. One of the features PB offers is loss-free order-preserving packet delivery, which is fundamental for high TCP throughput and critical to many web services [37]. We implement the LTE mobility management application using *only* PB APIs and basic SDN flow management APIs (i.e. FlowMod in OpenFlow). For comparison, we also implement two alternative solutions that do not guarantee loss-free order-preserving delivery: one buffers packet at the control-plane, one does not buffer any packet. The control-plane buffering solution is what current SDN frameworks would adopt due to lack of buffering abstractions for the data-plane; the no-buffering solution serves as a baseline with only TCP retransmissions.

We compare these solutions by simulating the simplest handoff scenario: one UE with an active TCP session detaches from one eNB and, after a set duration (50ms, typical LTE handoff duration [22]), reattaches to a new eNB where the TCP session is resumed (See Figure 4). The UE side (simulated wireless link) bandwidth constraint is 500Mbps[6], and the end-to-end latency is 20ms. Server runs an iPerf process. Consecutive handoffs are simulated with UE moving back and forth between two eNBs. Handoff intervals range from 1s to 2s, which corresponds to a handoff frequency from 0.5/s to 1/s. The results are shown in Figure 7. We calculate the optimal number by assuming no throughput recovery delay.

Finding 2. PB-enabled LTE Mobility Management yields near-optimal result for pedestrian and vehicular speeds in small cell environments.

6.3.2 Fast Mobility Management. As described in Section 5.1, the Fast Mobility Management application supports high mobility by multicasting at the anchor switch. We implement the FMM application with PB APIs and SDN flow APIs similar to LMM. As mentioned in Section 6.3.2, we implement a simple and efficient synchronization mechanism inside FMM's buffers. When the source buffer's TX mode vport is changed to RX (*Forwarding* to *Buffering*), a special packet containing the identification information will be sent to the target buffer and be used to purge duplicate packets. In microbenchmark this mechanism incurs less than 1ms delay for comparing and purging 10,000 packets, which is one order of magnitude smaller than the handoff durations. We carry out high-frequency handoff tests with the FMM application and calculate the average throughput under each setting. We selected 10ms as the handoff duration, and ten handoff intervals ranging from 10ms to 100ms. In this way, we can see how FMM performs when handoff duration is larger than handoff interval. These settings combined give us handoff frequencies ranging from less than 10 times/s to 50 times/s. Results are shown in Figure 8.

Finding 3. PB-enabled Fast Mobility Management solution delivers near-optimal throughput with 5% or less overhead for ultra high handoff frequencies.

6.3.3 Connectionless Communications. Both mobility management applications are data-plane intensive with mild control-plane workload. In contrast, connectionless communication (CC) application is less latency sensitive due to

[5]As explained in Section 3.4, PB's vport mode design allows applications to change buffer state by simply changing the mode of a vport.
[6]Despite the term LTE mobility management, this scheme will be integrated in the 5G infrastructure as 5G embraces different access technologies to ensure seamless user experience. Therefore 500Mbps is not an overkill.

its asynchronous nature and less throughput hungry, yet imposes much higher control-plane overhead. An overwhelming number of IoT devices generate periodic burst of traffic and requires the control-plane to handle these events efficiently. We evaluate this scenario by running simulations on a CC application that controls 1000 data sources and one data sink. To simplify the setting, we set up one PBS instance connected to 1000 virtual hosts working as "sensors" and another virtual host that periodically collects the "sensor" data. In each run, the "sensors" send an upload request to the application, which assigns a small buffer for each of them and installs the proper flow rules that direct the uplink traffic into each buffer. After all data has been collected and stored in the network, the server node sends a download request to the application, which changes the state of each buffer and directs all traffic toward the server node. We record the timestamp for each stage throughout the process, such as "upload request sent", "upload starts", "download finishes", on both end hosts and controller. For comparison, we also implement a control-plane solution that stores all "sensor" data on the controller. To see if data volume affects the result, we give each "sensor" two files to send: one's size is 10KB and the other 1MB.

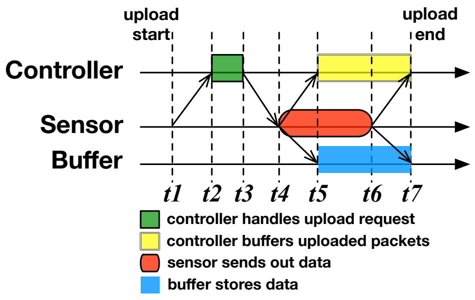

Figure 9: Upload process timeline

Figure 9 shows a timeline of the upload process in both solutions. For simplicity, here we assume both upload processes take the same time[7] to finish: *t7-t1*. For the CC application, because control and buffering are decoupled, the controller only handles one upload request and is occupied for *t3-t2* which is constant. In contrast, the control-plane buffering solution adds another *t7-t5* which grows linearly with file sizes.

In our simulations, each upload request takes around 0.53ms to process, while the control-plane buffering process takes 82ms for the 10KB file and 2 seconds for the 1MB file. The download request takes around 0.54ms to process, and the download process lasts 1ms and 85ms respectively.

Finding 4. PB-enabled Connectionless Communication application handles upload requests with constant overhead, outperforming control-plane buffering by 160 and 4000 times for 10KB and 1MB files respectively.

[7]In reality control-plane buffering takes much longer due to limited control plane bandwidth

7 DISCUSSION

PB's applicability outside 5G and API generality. Even though PB is strongly motivated by the use cases and visions of 5G, we believe that PB could potentially be useful in other scenarios as well, such as data center. The results shown in Section 6.2 also demonstrate this potential. The APIs are designed independently of the applications. In fact, two of the three use cases (Fast Mobility Management and Connectionless Communications) are developed after the APIs are finalized. We believe the APIs are fairly general as it captures the three critical elements in controlling flow buffering: What (flow matching), When (event triggered) and Where.

Limitations. PB is designed to be fully backward compatible with existing SDN applications, thus its deployment is independent of its utilization. That being said, PB might not always be effective due to its reliance on e.g. low control latency (controller proximity). Offloading partial control to PBS (e.g. changing buffer state locally based on pre-configured policy without involving the controller) could mitigate high control latency. We leave this for further study.

Security considerations. As user packets are kept indefinitely inside buffers, certain access controls need to be in place to protect user privacy. For example, applications should not have direct access to buffers&vports not created by them (unless they are recycled by the controller), and buffers should be emptied before reused. Malicious devices could also try to DDoS attack by tricking applications into allocating exceptionally large buffers. We believe there are both opportunities and challenges with Programmable Buffer in the security area, and we leave them for future explorations.

8 CONCLUSION

A new SDN-NFV solution is proposed to allow external controllers to manage the available memory on software switches for orchestrating where, when and how network flows are buffered. It is shown that the proposed abstractions can be applied to provide mobility management, as it is done in the current LTE networks, with near-optimal performance. PB significantly outperforms alternative SDN solution that uses controller for buffering purposes. PB is also backward compatible with existing SDN applications. Benchmarks on the prototype show great performance and scalability potentials of PB, for it exceeds 5G standards in every aspect we tested. Programmable buffers can easily deliver 90+ Gbps throughput with large packets, and scale out on Programmable Buffer Switches with acceptable overhead. Moreover, the PB abstraction is powerful enough to support fast mobility management that handles extreme mobility scenarios with less than 5% performance drop. Last but not least, the decoupling between control and buffering allows the PB control-plane to be scalable, as demonstrated in the connectionless communication use cases.

ACKNOWLEDGMENTS

This work was supported in part by NSF under award CNS-1345226, and by gift funding from Huawei Technologies, Inc. We would like to thank our shepherd, Radhika Niranjan Mysore, and the anonymous reviewers for their constructive and valuable feedback. We are also deeply grateful for Michio Honda's input on the paper and his assistance in adapting mSwitch and netmap for our benchmarks.

REFERENCES

[1] 5G Vision: 100 Billion Connections, 1 ms Latency, and 10 Gbps Throughput. http://www.huawei.com/minisite/5g/en/defining-5g.html. (Accessed on 02/26/2018).
[2] 5G Vision Brochure. https://5g-ppp.eu/wp-content/uploads/2015/02/5G-Vision-Brochure-v1.pdf. (Accessed on 02/26/2018).
[3] AT&T Unveils 5G Roadmap Including Trials In 2016 | AT&T. http://about.att.com/story/unveils_5g_roadmap_including_trials.html. (Accessed on 02/26/2018).
[4] Docker. https://www.docker.com/.
[5] Floodlight OpenFlow Controller. http://www.projectfloodlight.org/floodlight/.
[6] gRPC. https://grpc.io/.
[7] Internet Connection Speed Recommendations. https://help.netflix.com/en/node/306. (Accessed on 02/26/2018).
[8] M-CORD Open Source Reference Solution for 5G Mobile Wireless Networks. https://www.opennetworking.org/solutions/m-cord/.
[9] netmap. http://info.iet.unipi.it/~luigi/netmap/.
[10] ONOS - A new carrier-grade SDN network operating system designed for high availability, performance, scale-out. https://onosproject.org/.
[11] Open vSwitch. http://www.openvswitch.org/.
[12] OpenDaylight. https://www.opendaylight.org/.
[13] P4. https://p4.org/.
[14] Ryu SDN Framework. https://osrg.github.io/ryu/.
[15] SDN / OpenFlow / Message Layer / FlowMod | Flowgrammable. http://flowgrammable.org/sdn/openflow/message-layer/flowmod/. (Accessed on 02/26/2018).
[16] Service-Oriented 5G Core Networks. http://carrier.huawei.com/~/media/CNBG/Downloads/track/HeavyReadingWhitepaperServiceOriented5GCoreNetworks.pdf. (Accessed on 02/26/2018).
[17] TUN/TAP - Wikipedia. https://en.wikipedia.org/wiki/TUN/TAP.
[18] 3GPP TR 138 913 V14.2.0. Study on Scenarios and Requirements for Next Generation Acess Technologies, May 2017.
[19] J. G. Andrews, S. Buzzi, W. Choi, S. V. Hanly, A. Lozano, A. C. Soong, and J. C. Zhang. What will 5G be? *IEEE Journal on selected areas in communications*, 32(6):1065–1082, 2014.
[20] M. T. Arashloo, M. Ghobadi, J. Rexford, and D. Walker. HotCocoa: Hardware Congestion Control Abstractions. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 108–114. ACM, 2017.
[21] A. Banerjee et al. Scaling the LTE Control-Plane for Future Mobile Access. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2015.
[22] A. Elnashar and M. A. El-Saidny. Looking at LTE in practice: A performance analysis of the LTE system based on field test results. *IEEE Vehicular Technology Magazine*, 8(3):81–92, 2013.
[23] A. Gember-Jacobson and A. Akella. Improving the safety, scalability, and efficiency of network function state transfers. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, pages 43–48. ACM, 2015.
[24] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 163–174. ACM, 2014.
[25] D. Han, S. Shin, H. Cho, J.-M. Chung, D. Ok, and I. Hwang. Measurement and stochastic modeling of handover delay and interruption time of smartphone real-time applications on LTE networks. *IEEE Communications Magazine*, 53(3):173–181, 2015.
[26] M. Honda, F. Huici, G. Lettieri, and L. Rizzo. mSwitch: a highly-scalable, modular software switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 1. ACM, 2015.
[27] M. Irland. Buffer management in a packet switch. *IEEE transactions on Communications*, 26(3):328–337, 1978.
[28] X. Jin, L. E. Li, L. Vanbever, and J. Rexford. Softcell: Scalable and flexible cellular core network architecture. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 163–174. ACM, 2013.
[29] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. 2017.
[30] R. P. Jover and I. Murynets. Connection-less communication of IoT devices over LTE mobile networks. In *Proc. of IEEE SECON'15*, pages 247–255. IEEE, 2015.
[31] K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S. Nikolenko, A. V. Sirotkin, and P. Eugster. A Programmable Buffer Management Platform. 2017.
[32] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.
[33] M. Moradi, L. E. Li, and Z. M. Mao. SoftMoW: A dynamic and scalable software defined architecture for cellular WANs. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 201–202. ACM, 2014.
[34] M. Moradi, F. Qian, Q. Xu, Z. M. Mao, D. Bethea, and M. K. Reiter. Caesar: High-speed and memory-efficient forwarding engine for future Internet architecture. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems*, pages 171–182. IEEE Computer Society, 2015.
[35] M. Moradi, W. Wu, L. E. Li, and Z. M. Mao. SoftMoW: Recursive and reconfigurable cellular WAN architecture. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 377–390. ACM, 2014.
[36] NGMN Alliance. 5G White Paper. *Next Generation Mobile Networks, White Paper*, 2015.
[37] B. Nguyen, A. Banerjee, V. Gopalakrishnan, S. Kasera, S. Lee, A. Shaikh, and J. Van der Merwe. Towards understanding TCP performance on LTE/EPC mobile networks. In *Proceedings of the 4th workshop on All things cellular: operations, applications, & challenges*, pages 41–46. ACM, 2014.
[38] J. S. Plank, M. Beck, W. R. Elwasif, T. Moore, M. Swany, and R. Wolski. The internet backplane protocol: Storage in the network. In *In Proceedings of the Network Storage Symposium*. Citeseer, 1999.
[39] Z. A. Qazi et al. KLEIN: A Minimally Disruptive Design for an Elastic Cellular Core. In *Proceedings of the 2nd ACM SIGCOMM Symposium on Software Defined Networking Research*, 2016.
[40] W. Roh, J.-Y. Seol, J. Park, B. Lee, J. Lee, Y. Kim, J. Cho, K. Cheun, and F. Aryanfar. Millimeter-wave beamforming as an enabling technology for 5G cellular communications: Theoretical feasibility and prototype results. *IEEE Communications Magazine*, 52(2):106–113, 2014.

[41] I. Seskar, K. Nagaraja, S. Nelson, and D. Raychaudhuri. Mobilityfirst future internet architecture project. In *Proceedings of the 7th Asian Internet Engineering Conference*, pages 1–3. ACM, 2011.

[42] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable Packet Scheduling at Line Rate. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, pages 44–57. ACM, 2016.

[43] S. Sur, V. Venkateswaran, X. Zhang, and P. Ramanathan. 60 ghz indoor networking through flexible beams: A link-level profiling. *SIGMETRICS Perform. Eval. Rev.*, 43(1):71–84, June 2015.

[44] Y. Wang, G. Xie, Z. Li, P. He, and K. Salamatian. Transparent flow migration for nfv. In *Network Protocols (ICNP), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.

[45] Y. Xu, Z. Wang, W. K. Leong, and B. Leong. An end-to-end measurement study of modern cellular data networks. In *International Conference on Passive and Active Network Measurement*, pages 34–45. Springer, 2014.

[46] V. Yazici, U. C. Kozat, and M. O. Sunay. A new control plane for 5G network architecture with a case study on unified handoff, mobility, and routing management. *IEEE Communications Magazine*, 52(11):76–85, Nov 2014.