A P4-based 5G User Plane Function

Submission Id: 33

ABSTRACT

The demands on mobile networks are constantly evolving, but designing and integrating new high-speed packet processing remains a challenge due to the complexity of requirements and opacity of protocol specifications. 5G data planes should be implemented in programmable hardware for both speed and flexibility, and extending or replacing these data planes should be painless. In this paper we implement the 5G data plane using two P4 programs: one that acts as a open-source model data plane to simplify the interface with the control plane, and one to run efficiently on hardware switches to minimize latency and maximize bandwidth. The model data plane enables testing changes made to the control plane before integrating with a performant data plane, and vice versa. The hardware data plane implements the fast path for device traffic, and makes use of microservices to implement functions that high-speed switch hardware cannot do. Our data plane implementation is currently in limited deployment on three university campuses where it is enabling new research on mobile networks.

1 INTRODUCTION

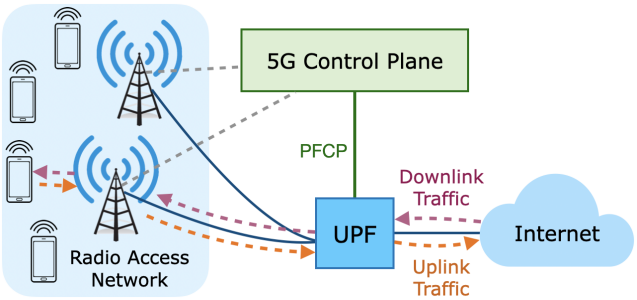

Figure 1: UPF is configured by the control plane via PFCP, a niche protocol specific to cellular networks. The UPF routes traffic destined for user devices (i.e., downlink traffic) towards the correct base station, and acts as a gateway for internet traffic sent by user devices (i.e., uplink traffic).

The emergence of 5G promises high speed and low latency, enabling a wide range of innovative applications like Internet of Things (IoT) and augmented/virtual reality. As a result, many organizations—from global carriers and cloud providers to university campuses and small businesses—want to deploy their own 5G networks, and customize them for their users and applications. Historically, mobile network technology has been closed and vertically integrated, making customization frustratingly difficult, if not impossible. Fortunately, this is changing with 5G due to the availability of open-source mobile core implementations, including Aether [6], MagmaCore [8], OpenRAN [10], and Free5GC [7], as well as the adoption of software-defined networking and cloud services in these platforms.

The heart of the 5G data plane is the User-Plane Function (UPF), as shown in Figure 1. The UPF not only serves as a full-fledged IP router, but also routes traffic to mobile devices as they move between base stations, buffers traffic for idle devices, enforces QoS constraints, accounts for subscriber usage, and more. The UPF must execute these features at ever-growing speeds for an ever-increasing number of mobile devices.

The UPF interacts with a complex control plane consisting of many components with different responsibilities (e.g., authentication, billing, etc.), where the control interface is defined by the Packet Forwarding Control Protocol (PFCP). At the same time, the set of features the UPF needs to support continuously evolves, due in part to specification changes, but also in response to deployments that require customization and new features. As a result, designers of Mobile Core platforms face the dual challenges of (i) implementing a sophisticated control plane, and (ii) implementing an extensible, high-speed UPF data plane.

This paper argues that specifying UPF functionality in the P4 language can help address both of these challenges. To this end, we present two P4-based UPF designs. The first, *model UPF*, defines the PFCP interface as a series of match-action tables, with match keys based on packet metadata and actions that process the packet. Here, the P4 program distills the essence of PFCP from a complex and evolving standards document [4]. With the model UPF data plane specified in P4, a P4 compiler can generate an RPC-based PFCP interface automatically, simplifying the process of implementing the control plane. The model UPF is also a functional UPF, suitable for running correctness tests on the control-plane implementation. Running on a software switch, the model UPF supports developing control-plane software in emulation environments like Mininet [9], without developers requiring access to special-purpose data-plane hardware. Additionally, recent research focused on automatically generating test cases based on P4 programs would allow the model UPF to be used to test other UPF implementations [15, 20].

The model UPF P4 program serves as a useful starting point for creating full-fledged UPF implementations for specific hardware targets. This leads to our second design, the *performant UPF*, which runs on the Intel Tofino programmable data-plane switch. While earlier open-source 5G platforms implemented the UPF entirely in software, hardware network interface cards (NICs) and switches offer higher speed and lower power. The performant UPF P4 program must grapple with the realities of limited data-plane resources. Some match-action tables in the model UPF are too large, requiring optimizations that break “wide" match-action tables into a collection of smaller tables. Other capabilities of a performant UPF cannot be expressed in P4, or cannot be supported in high-speed packet-processing hardware at all. Here, we rely on microservices to support certain functionality, such as buffering traffic for idle mobile devices. The end result is an efficient system with a hardware data plane processing most traffic, an a set of microservices handling other UPF functionality.

Rule	Rule Key(s)	Rule Parameters
Packet Detection Rule	IP Address, 5-Tuple, Tunnel Headers, *Endpoint DNS Name Regex*	FAR-ID, QER-ID, URR-ID, Decapsulation Flag
Forwarding Action Rule	FAR-ID	(Forward, Buffer, Notify) Flags, Tunnel Headers (optional), BAR-ID (optional)
Buffering Action Rule	BAR-ID	*Buffer Depth, Buffer Duration*
Usage Reporting Rule	URR-ID	Counter Index, *Reporting Frequency or Threshold*
QoS Enforcement Rule	QER-ID	*QoS Flow ID (QFI), Guaranteed BitRate, Maximum BitRate*

Table 1: The rules a 5G control plane uses to configure a UPF, the match keys used to look up a rule, and the parameters loaded into a packet's metadata by said rule. Italicized fields are either scaffolded or not present in the model UPF.

Many existing works aim to solve issues present in both LTE and 5G such as control signalling load [18, 21, 23], fault tolerance [13, 19], and software data-plane performance [22] but few seek to implement the data-plane in hardware, although there are proprietary P4-based solutions [2]. TurboEPC [23] implements LTE's equivalent of the UPF in P4 switches and reduces control-plane load by processing some common control messages in the data-plane, but it requires control plane modification and does not discuss features like idle buffering that are currently unsupported by P4. Aghdai et. al [12] introduce a new P4-based network function to the mobile core for reducing latency between user devices and edge services, but they do not implement the UPF or its LTE equivalent.

The remainder of the paper is organized as follows. §2 presents background on the User Plane Function and the control-plane interface. §3 presents the model UPF, including how to synthesize the control-plane interface and specify the data plane. (We will release the model UPF P4 program as open source, to serve as executable UPF documentation for the community.) §4 describes the performant UPF for the Intel Tofino switch, including how to work within limited data-plane memory and interface to an external buffering microservice. §5 presents our experiences with the two UPFs in an open-source mobile core platform (including deployments on university campuses), and §6 discusses future research directions.

2 USER PLANE FUNCTION (UPF)

The User Plane Function (UPF) connects the base stations of the Radio Access Network (RAN) to the Internet. It performs packet processing for user devices, including supporting mobility, buffering for idle devices, traffic accounting, and quality-of-service based on rules configured by the control plane, as summarized in Table 1.

Traffic classification: Each packet corresponds to a user device attached to the cellular network. The UPF associates a packet with the corresponding user device and traffic class, based on Packet Detection Rules (PDRs). A PDR may simply match the device's IP address, or consider tunnel headers, the five-tuple, or even the domain name of the other end-point. The matching PDR determines how other parts of the UPF process the packet. Each attached user device has at least two PDRs, for uplink and downlink traffic, respectively, and possibly more to support multiple traffic classes (e.g., for different QoS levels, pricing plans, etc.). The control plane installs, changes, and removes PDRs when a device attaches, moves to another base station, and detaches, respectively.

Mobility and packet forwarding: User devices connect to new base stations as the user moves. To tunnel downlink packets to the right base station, the UPF applies a Forwarding Action Rule (FAR) identified by the PDR during packet classification. The FAR for downlink traffic indicates the tunnel header field and the IP address of the base station. More generally, a FAR specifies a set of actions (using flags) to apply to the packet, including tunneling, forwarding, buffering, and notifying the control plane. For example, a typical FAR for uplink traffic contains only a 'forward' flag, signifying that the packet is permitted to enter the UPF's IP router functionality. In contrast, the 'notify' flag sends an alert to the control plane to wake an idle device. FARs are installed and removed when a device attaches or detaches, respectively, and the downlink FAR changes when the device moves, goes idle, or wakes.

Buffering for idle devices: When a user device goes idle, the UPF buffers downlink traffic that arrives for that device until it reawakens; this feature is increasingly important as battery optimizations and limited radio spectrum push devices to spend more time idle. When traffic first arrives, the UPF sends an "Downlink Data Notification" alert to the control plane, which triggers the base station to attempt to wake the device. Once the device awakens, the UPF releases the buffered traffic and resumes normal forwarding. The buffering and notification functions are activated by modifying a FAR to include 'buffer' and 'notify' flags. An additional set of Buffering Action Rules (BARs) decide settings like the maximum number of packets (and the maximum duration) to buffer. The identifier of the BAR to use is determined by the FAR that triggered buffering. If no BAR is provided, default settings are assumed.

Traffic accounting: The UPF sends usage reports for each user device to the control plane. These reports include counts of the packets sent and received by the UPF for both the uplink and downlink traffic for each user device and traffic class. Service providers use these measurements to limit and bill their customers separately for upload and download usage. The control plane installs and removes Usage Reporting Rules (URRs) when the device attaches and detaches, respectively. Each URR includes parameters specifying whether usage reports should be sent periodically or when a quota is exceeded. Typically a device has two URRs, one for uplink and downlink usage, respectively. If a user's plan includes special treatment for certain types of traffic, an additional URR is created for each traffic class (e.g., to support plans with free VoIP/video calls).

Quality-of-Service: The UPF guarantees a minimum amount of available bandwidth and enforces a bandwidth cap for each user device, for uplink and downlink packets for each traffic class. These parameters are decided by per-device Quality Enforcement Rules (QERs). The identifier of the associated QER is determined by the PDR in the traffic-classification process. The control plane installs and removes QERs when a device attaches and detaches, respectively, and are modified according to operator-defined events such as when the network becomes more or less congested, the user device exceeds a quota, or the network policy changes (e.g., the user signs up for a new pricing plan). The UPF can perform traffic policing to enforce the bandwidth cap, as well as packet scheduling to ensure a minimum bandwidth in conjunction with admission control in the control plane.

3 THE MODEL UPF

The PFCP protocol used for communication between the control plane and the UPF can be difficult to understand, even though the rules it installs are actually simple match-action rules, as summarized in Table 1. Additionally, documentation on the operations applied to a packet by the UPF and the order in which they apply are scattered across the 5G specifications. To address both of these issues, we propose *modeling* the UPF with a P4 program. Such a model would provide unambiguous, executable documentation on the UPF and, with the addition of P4Runtime, provides a simple control-plane interface as well. More specifically, we have two objectives in the implementation of a model UPF:

(1) The model UPF should serve as an interface that the 5G control plane can expect a data-plane implementation to expose. Developers creating their own data plane need only present the same interface as the model UPF in order to integrate with the control plane. Implementing the same interface also means that testing infrastructure can be reused to verify the behavior of a new UPF.
(2) The model UPF should implement a minimally functional UPF, to act as a valid and executable data plane suitable for running correctness tests on the 5G control plane in the absence of a more sophisticated data-plane implementation. The implemented functions can also serve as behavioral references for implementers of new UPFs.

3.1 Synthesizing the Control-Plane Interface

P4 is a language for specifying packet-processing pipelines, but it does not provide any means for the control plane to configure those pipelines at runtime. P4Runtime fills this need. P4Runtime is an RPC (Remote Procedure Call) protocol that allows a P4Runtime client running in the control plane to read and write table entries, read counter values, and install new P4 programs from a P4Runtime server running on a data plane [1]. The format of table and counter reads and writes is determined by a `p4info` file, akin to a header file, that is generated when a P4 program is compiled. For example, for each table in a P4 program, the `p4info` contains a description of the table name, match keys, action names, and action parameters, but not details of the operations or packet modifications performed by the table. A P4Runtime client can install an entry into a table by sending a write request message containing a table entry matching the format specified in the `p4info`. If we create P4 representations of every UPF feature, P4Runtime automatically gives us a control interface to those representations.

To implement a real UPF that uses the model UPF interface, the implementer must only implement a P4Runtime server that accepts messages matching the `p4info` of the model UPF, as shown in Figure 3. Since P4Runtime is based on gRPC, client and server stub implementations can be automatically generated from the P4Runtime protocol specification for a variety of popular languages. The stub implementations take care of message serialization and connection management; the implementer need only worry about reading and writing message objects, accelerating the development and integration of new UPFs. However, to support these new UPFs, the 5G control plane must speak P4Runtime instead of PFCP. This is handled by the implementation of a PFCP-to-P4Runtime translation microservice, which must be done once and can be reused by different control plane implementations without modification.

An important note here is that the data plane driven by the P4Runtime server need not exactly adhere to the P4 program presented to the client. For example, the data plane may consist of two parallel ASICs, each with one match-action table, but the P4Runtime server may choose to abstract them as a single device with a single table. In such a setting where the data plane does not match the `p4info` presented to clients, it is up to the server to translate reads/writes to the representative P4 program into reads/writes to the true data plane. Although this is translation must be manually implemented, there is research interest in automating such translations [15].

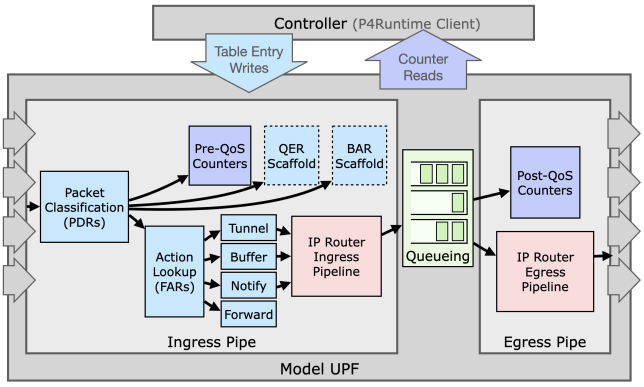

Figure 2: The model UPF P4 pipeline

3.2 Specifying the Model UPF Data Plane

In order to meet objective (1), we need a P4 program that, when compiled, generates a set of P4Runtime messages that directly map to each type of rule the 5G control plane may install. To that end, we start with a simple table for each rule type, with the match keys and action parameters of each table given in Table 1. The P4 actions in each table initially only load the described action parameters into a packet's metadata. With the exception of the PDR table, each table's match key is just a rule identifier. The PDR table performs more complex matching, being capable of ternary matching on packet header fields. Although the 5G specifications [4] permit matching on endpoint DNS names, this is challenging to support in P4 and we leave it for future work. We use these simple tables as a starting point for the model UPF that meets objective (1).

The secondary goal of our model UPF is to implement enough of these tables to create a minimally functional UPF. The PDR table can be made functional by making every other table match on the rule identifiers loaded by the PDR table. As a result, each other table in the model depends upon the PDR table. As seen in Figure 2, the FAR table is made functional with the addition of tables that read and act upon the loaded action flags. Each action requires its own table because the actions are not mutually exclusive. Since

[1] The server is not exactly "on" the data plane. Modern switches are typically comprised of a both a CPU and a packet-processing ASIC. The ASIC is configured via the PCI bus by control processes (like a P4Runtime server) running on the CPU.

these tables only read a single flag and perform a single action, they each have only one hardcoded entry. Dropping, forwarding, and encapsulation are easy to do, and the 'notify' action can be implemented by using the P4 *packet digests* feature to send a short message to the control plane.

Per-device usage tracking is implemented with arrays of packet counters in the model UPF's ingress and egress pipelines (i.e., before and after QoS enforcement), indexed by the counter index loaded in the URR table. However, usage reports containing values from these counters cannot be periodically produced by a P4 data plane because P4Runtime only permits the *pulling* of counter values by the control plane, not *pushing* by the data plane. This is resolved by tasking the PFCP-to-P4RT translator with periodically polling the model UPF's counters to generate usage reports. As a consequence of moving report generation to the controller, the URR table does not need to contain usage reporting parameters.

There are some features left only as non-functional placeholder tables in the model UPF. We refer to these tables that only satisfy objective (1) as *scaffolding*. The first feature scaffolded is buffering, because currently available P4 switches cannot indefinitely buffer packets. If a device is idle, the model UPF simply drops packets destined for that device—simulating a UPF with no available buffering memory. The second feature scaffolded is QoS enforcement, which consists of the enforcement of minimum and maximum bitrates. We settle for simple FIFO packet scheduling because verifying forwarding behavior, not QoS properties, is the primary goal of processing packets with the model UPF. We leave implementing bitrate restrictions in P4 to future work.

With the features implemented so far, the model UPF can forward packets between the RAN and the mobile core. In our 5G ecosystem this model UPF only processes packets for the purposes of verifying forwarding behavior, but it could be used for serving real user devices, albeit with poor performance for several reasons: (1) The FIFO scheduling of device traffic does not impact correctness, but devices will not be guaranteed bandwidth. (2) Zero buffering capacity means that packets sent when a device is idle must be retransmitted, adding some delay. (3) It is designed for and runs on a software switch. It could be compiled for a hardware switch, but the design is intentionally simple and would not make efficient use of hardware resources, which is an issue we address in the next section.

4 THE PERFORMANT UPF

Previously, we discussed how UPF features are either (1) implementable in the model UPF, but are done inefficiently for the sake of simplicity, or (2) not implementable in P4 and are only expressed as scaffolding in the model UPF. In this section we discuss how these two types of features are efficiently implemented using hardware switches and software microservices. Our two primary contributions are an extension to an established hardware switch P4 program for handling features of type (1), and a control-plane application for managing table entries in the extension, talking to other microservices for features of type (2), and hosting a P4Runtime server that represents the model UPF. The server receives reads and writes intended for the model UPF, and it is up to the control application to translate those reads and writes to physical switch reads and writes for (1) and additional microservice calls for (2).

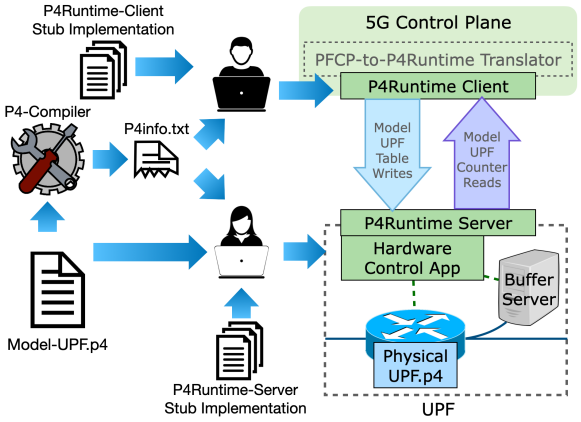

Figure 3: How the model UPF is used to develop and deploy a real UPF. A P4 compiler takes the model program and produces a p4info file, akin to a header file for the P4 program. A network operator uses the p4info and a P4Runtime client framework to implement a translator for the control plane. A researcher references the model and uses the p4info and the P4Runtime server framework to implement a full UPF.

4.1 Dealing with Limited Data-Plane Memory

Although some parts of the model UPF can be directly implemented in the physical switch, it would not be wise to do so in all cases. For example, the PDR table in the model UPF is written very inefficiently for the sake of clarity and flexibility; it permits highly complex rules that require large amounts of costly switch TCAM. To maximize the number of rules that may fit in a single physical switch, we note that the majority of PDRs fall into one of two patterns: (1) simple uplink rules that only match on the tunnel destination and identifier[2], and (2) simple downlink rules that only match on the device IP address. We use this observation to create three parallel PDR tables in the physical switch: one for common-case uplink rules that exactly matches (using the relatively plentiful switch SRAM instead of TCAM) on two tunnel header fields, one for common-case downlink rules that exactly matches on only the IP destination, and one small, inefficient table that is as flexible as the model PDR table and will only be used for rare, complex rules. The control app is responsible for mapping model PDR table entries to the most suitable hardware table. If more PDR patterns are observed in the future, additional tables optimized for each pattern can be created.

4.2 Interfaces to Buffering Microservice

When packets arrive from the internet destined for an idle device, the UPF should buffer packets on behalf of that device and send an alert to the 5G control plane to awaken it. Currently available hardware P4 switches are designed for data centers and thus do not have

[2]The tunneling protocol used in mobile networks for sending packets between the UPF and base stations, GTP, includes a header field called the Tunnel Endpoint Identifier (TEID), which can be used by PDRs to map packets to devices without parsing inner headers.

large buffers or the ability to hold packets indefinitely. To cover this feature gap, we use a buffering microservice, controllable via gRPC, that is provided by our chosen 5G ecosystem. The microservice indefinitely holds any packets that it receives, and releases them back into the network when a control signal is received via gRPC.

When a device goes idle, the control plane installs a FAR in the model UPF with the 'buffer' flag set. The control app translates this model FAR entry to flow rules in the physical UPF P4 module that redirect packets destined for the idle device to the buffering microservice. Packets are redirected to the microservice without modifying the IP headers by placing them in a tunnel. The tunneling protocol used for sending data to the microservice is the same as that used to send packets to 5G base stations, which allows the hardware switch and control app to treat the buffering microservice as just another base station. Additionally, not implementing another tunnel protocol in the switches reduces resource usage in the packet parser and deparsers, which is an important concern for industrial network deployments that have continuously growing feature lists.

When the first packet of a new flow arrives at the buffering microservice, the microservice sends an alert to any currently connected gRPC clients that a new flow is being buffered. The control app listens for this event, and translates it to an alert to the 5G control plane that the device corresponding to that flow should wake up. The alert is sent as a 'packet digest' emanating from the model UPF. When a device wakes up, the control plane modifies the FAR installed in the model UPF by unsetting the 'buffer' flag and setting the 'tunnel' flag. When this specific transition occurs, the control app sends a gRPC signal to the buffering microservice, instructing it to release all packets for that device back to the physical switch. Packets arriving at the physical switch from the buffering microservice skip the portion of the UPF module they encountered before buffering, to give the illusion of being buffered in the middle of the switch. Their processing resumes at the tunneling stage, where they are encapsulated and routed to the appropriate base station.

4.3 QoS and Usage Reporting

QERs cannot be implemented in the model UPF because P4 does not support the expression of packet schedulers. However, currently available hardware P4 switches have simple schedulers with programmable weights and priorities, programmed using runtime interfaces unrelated to P4. It may be possible to approximately enforce bitrate guarantees and limits using this scheduler with an approach like Rotating Strict Priority [24], but we leave that for future work and for now simply map the QoS class identifiers loaded by the QERs to one of the available queues.

Usage counters for generating Usage Reporting Rules are implemented in the hardware switches identically to how they appear in the model UPF, with counter arrays in the switches' ingress and egress. When the PFCP-to-P4 translation microservice requests counter values from the model UPF, the control application simply polls the hardware switch counters and relays the response.

5 DEPLOYMENT EXPERIENCE

Our prototype has been included in a popular 5G mobile core ecosystem and is currently in limited deployment on three university campuses, where it is being used for research on improving the reliability and performance of mobile networks.

5.1 Prototype Implementation Details

The hardware control application (see Figure 3) responsible for translation between the model and hardware UPFs is written as a Java application for a network controller OS [11] and consists of ~5,200 lines of code. The Model UPF is written in ~760 lines of P4 code for the Behavioral Model version 2 (BMv2) software switch. The hardware UPF was written in ~400 lines of P4 code as an extension for an established leaf-spine fabric P4 switch program [5]. In our campus deployment, the performant UPF is installed in the leaf of a 2x2 leaf-spine topology of 6.4 Tbps Edgecore Wedge100BF-32X and 32QS switches that use the Intel Tofino ASIC. The leaf-spine topology connects a RAN of small cells to compute nodes and the internet.

Currently, the hardware UPF pipeline uses less than 15% of the Tofino chip's total available SRAM memory to support 17,500 user devices. The primary sources for the SRAM usage are the PDR tables, FAR tables, and URR counters, each of which must have capacity equal to twice the maximum number of connected devices (since each device has at least two of each rule—one uplink and one downlink). The program has not been aggressively optimized and achieving support for several times more devices is likely, although memory will still become a limitation long before software switch scales are reached. Potential directions for supporting more devices in the face of memory limitations are discussed in Section 6.

5.2 Testing of the UPFs

For unit testing of both the UPF programs, we use the Python-based Packet Testing Framework (PTF). In a PTF unit test, a P4 program is installed in a software switch, a fixed set of P4Runtime messages are sent to the switch to populate the tables, and then packets are sent through the switch. The test is passed if packets output by the switch adhere to expectations. In our setting, the packets injected match the format of data traffic to and from a 5G RAN. There are unit tests for detached, attached, and idle device states. For the model program, the software switch used is BMv2. For the hardware program, it is the Intel Tofino ASIC simulator provided by the Tofino SDK. Functional equivalence between the model and hardware UPF programs is empirically checked by writing unit tests that inject and verify the same input and output packets in both programs. There are currently ~550 lines of PTF-based unit tests for the model UPF, and ~2600 lines of tests for the performant UPF, although many of these tests cover the non-UPF modules. If more robust equivalence tests are needed, projects like Avenir [15] and p4pktgen [20] may be used in the future.

For tests more intensive than unit tests, integration tests are implemented using Mininet and Docker containers, shown in Figure 4. One container runs a Mininet topology with hosts for an internet endpoint, a base station, and a buffering server, with one leaf switch operates as the performant UPF. There are also containers for the control app, the PFCP-to-P4Runtime translator, and a mock 5G control plane that injects control messages. To reduce the number of moving parts in a test, the Mininet topology and control app can

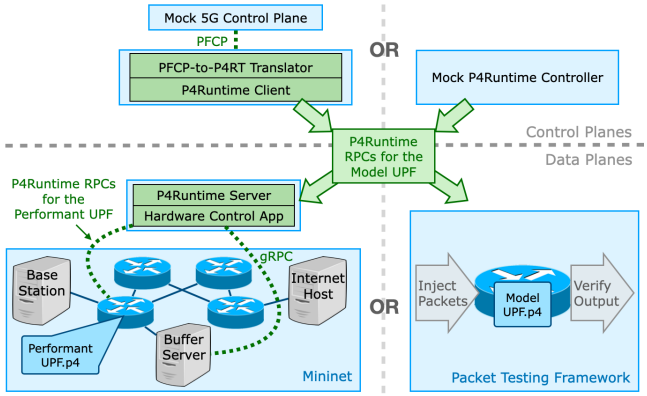

Figure 4: The topology of containers used for integration testing. Each blue box is a separate container. There are two possible control plane and two possible data planes, and all four pairings are valid tests.

be replaced by the model UPF, and the mock 5G control plane can be replaced by a simple mock P4Runtime controller.

5.3 UPF Extensibility

To show that our P4-based 5G data plane can be extended and used to support novel research, we extended the performant UPF pipeline with publicly available P4 code [3, 16] for detecting aggressive flows during times of congestion using a data-plane sketch, installed in the hardware switch memory unused by the UPF module. When congestion builds and an aggressive flow is detected, the control plane installs temporary block rules for those flows to prevent benign flows from experiencing excessive drops. The performant UPF pipeline modified with this sketch was installed on the leaf of a 2x2 leaf-spine topology in a campus deployment, and aggressive traffic was sent between servers connected to the leaf to simulate an attack on the UPF itself. Benign traffic was sent by mobile devices connected to a small cell attached to the leaf node and experienced service interruptions before the sketch was activated. No noticeable drops in service were observed after alerts were activated.

6 NEXT STEPS

There are several possible directions this project can take in the near future to address hardware memory limitations. Although the hardware switches used in our deployments run at terabits per second and are thus easily able to achieve 5G speeds, they do not have sufficient memory to maintain connection state for millions of simultaneously connected devices that a UPF is expected to support. Each connected device requires state in the UPF for match-action table entries for PDRs, FARs, and counters. However, it may be possible to compress, approximate, or offload this state to reduce memory usage.

Approximate usage reporting: For exact usage reporting, the data plane maintains four counters for each device: two for uplink (before and after QoS enforcement), and two for downlink. However, usage reporting is primarily intended to detect devices that exceed usage quotas. If only a small fraction of devices come close to or exceed their quotas, then it may be possible to replace exact counters with heavy-hitter sketches like count-min sketch [17], or heavy-hitter caches like PRECISION or HashPipe [14, 25]. Such data structures greatly reducing the data-plane memory used at the cost of some error in the final results. The absence of false positives in heavy-hitter caches makes them more suitable for usage reporting than sketches like CMS, as no subscribers would be falsely reported as exceeding their quotas.

Pseudo-random tunnel identifiers: For downlink traffic, the UPF must have match-action tables for every device that match on the device's IP address and load the correct Tunnel Endpoint Identifier (TEID) to use when encapsulating the packet. The TEID is a 32-bit field that is chosen at random by the control plane in an effort to defend against certain brute-force attacks [1]. The IP-to-TEID mapping requires at least 64 bits for each device, and cannot be compressed due to its randomness. However, if the control plane were modified to instead generate the TEID from the device IP address using a cryptographically secure pseudo-random number generator (RNG), the UPF data plane would only need to store the parameters of this RNG to generate the correct TEID for every packet. This approach has the downside that downlink packets for invalid IP addresses are not filtered, so it must be paired with an allowlist. Making an allowlist that is more memory-efficient than simply storing all valid device IP addresses is one challenge. Handling the case where devices each have more than one downlink tunnel (for flows with different QoS classes) is another challenge.

Parallel hardware and software UPFs: Although there may be millions of simultaneously connected devices sending traffic through the UPF, not all of them require high bandwidth. IoT devices and low-power sensors may send infrequently or at extremely low bitrates (tens of bits per second) and thus do not need hardware speeds. A software switch could easily handle millions of such connections, since software switches have access to orders of magnitude more memory than hardware switches. A UPF architecture that capitalizes on this disparity could maintain two parallel data planes: one running in a hardware switch and one in software. When devices first connect, their connection state is first placed in the software data plane. If the device begins sending or receiving traffic at a high rate, its connection state is moved to the hardware data plane. Detecting high-bandwidth flows could be done in the network using heavy-hitter sketches or similar data-plane structures [14, 16], and the dual data plane could be hidden from the control plane by the high-level interfaces defined by the model UPF.

Accurate QoS enforcement: The 5G control plane installs at least one QoS Enforcement Rule (QER) for every connected device consisting of a Maximum Bandwidth and a Guaranteed Bandwidth. If a device has multiple QoS traffic classes, one QER is installed per class. Ideally this would be implemented using a Hierarchical QoS (HQoS) scheduler, with a separate queue for every (device, QoS class) pair. Unfortunately, currently available P4 switches only have tens of queues per port, far short of the tens to hundreds of thousands needed by HQoS. One possible research direction would be to either approximate a HQoS scheduler or some other appropriately fair scheduler (like Rotating Strict Priority [24]) in P4 using a combination of the available queues, registers, meters, and other P4 features.

REFERENCES

[1] 2017. *Threats to Packet Core Security of P4 Networks.* Retrieved June 9th, 2021 from https://positive-tech.com/expert-lab/research/epc-research/

[2] 2019. *The Kaloom 5G User Plane Function.* Retrieved June 7th, 2021 from https://www.mbuzzeurope.com/wp-content/uploads/2020/02/Product-Brief-Kaloom-5G-UPF-v1.0.pdf

[3] 2020. *ConQuest Github Repo.* Retrieved June 15th, 2021 from https://github.com/Princeton-Cabernet/p4-projects/tree/master/ConQuest-tofino

[4] 2020. *Interface between the Control Plane and the User Plane nodes.* Retrieved April 3rd, 2020 from https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3111

[5] 2020. *The ONOS fabric.p4 switch.* Retrieved June 16th, 2021 from https://github.com/opennetworkinglab/onos/blob/master/pipelines/fabric/impl/src/main/resources/fabric.p4

[6] 2021. *Aether Project.* Retrieved June 6th, 2021 from https://aetherproject.org

[7] 2021. *Free5GC.* Retrieved June 6th, 2021 from https://www.free5gc.org

[8] 2021. *Magma Core.* Retrieved June 6th, 2021 from https://www.magmacore.org

[9] 2021. *Mininet.* Retrieved June 10th, 2021 from http://mininet.org

[10] 2021. *O-RAN Alliance.* Retrieved June 6th, 2021 from https://www.o-ran.org

[11] 2021. *ONOS Github Repo.* Retrieved June 16th, 2021 from https://github.com/opennetworkinglab/onos

[12] Ashkan Aghdai, Mark Huang, David Dai, Yang Xu, and Jonathan Chao. 2018. Transparent Edge Gateway for Mobile Networks. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. 412–417. https://doi.org/10.1109/ICNP.2018.00057

[13] Mukhtiar Ahmad, Syed Usman Jafri, Azam Ikram, Wasiq Noor Ahmad Qasmi, Muhammad Ali Nawazish, Zartash Afzal Uzmi, and Zafar Ayyub Qazi. 2020. A Low Latency and Consistent Cellular Control Plane. In *ACM SIGCOMM*. 648–661.

[14] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2020. Designing Heavy-Hitter Detection Algorithms for Programmable Switches. *IEEE/ACM Transactions on Networking* 28, 3 (2020), 1172–1185. https://doi.org/10.1109/TNET.2020.2982739

[15] Eric Hayden Campbell, William T. Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. 2021. Avenir: Managing Data Plane Diversity with Control Plane Synthesis. In *USENIX Symposium on Networked Systems Design and Implementation*. 133–153.

[16] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. 2019. Fine-Grained Queue Measurement in the Data Plane. In *ACM SIGCOMM Conference on Emerging Networking Experiments And Technologies*. 15–29.

[17] Graham Cormode and S. Muthukrishnan. 2005. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *J. Algorithms* 55, 1 (April 2005), 58–75. https://doi.org/10.1016/j.jalgor.2003.12.001

[18] Ali Mohammadkhan, K.K. Ramakrishnan, Ashok Sunder Rajan, and Christian Maciocco. 2016. CleanG: A Clean-Slate EPC Architecture and ControlPlane Protocol for Next Generation Cellular Networks. In *ACM Workshop on Cloud-Assisted Networking*. 31–36.

[19] Vasudevan Nagendra, Arani Bhattacharya, Anshul Gandhi, and Samir R. Das. 2019. MMLite: A Scalable and Resource Efficient Control Plane for Next Generation Cellular Packet Core. In *ACM Symposium on SDN Research*. 69–83.

[20] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. 2018. P4pktgen: Automated Test Case Generation for P4 Programs. In *ACM Symposium on SDN Research*.

[21] Matteo Pozza, Ashwin Rao, Armir Bujari, Hannu Flinck, Claudio E. Palazzi, and Sasu Tarkoma. 2017. A refactoring approach for optimizing mobile networks. In *IEEE International Conference on Communications*. 1–6.

[22] Zafar Ayyub Qazi, Melvin Walls, Aurojit Panda, Vyas Sekar, Sylvia Ratnasamy, and Scott Shenker. 2017. A High Performance Packet Core for Next Generation Cellular Networks. In *ACM SIGCOMM*. 348–361.

[23] Rinku Shah, Vikas Kumar, Mythili Vutukuru, and Purushottam Kulkarni. 2020. TurboEPC: Leveraging Dataplane Programmability to Accelerate the Mobile Packet Core. In *ACM Symposium on SDN Research*. 83–95.

[24] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *USENIX Symposium on Networked Systems Design and Implementation*. 1–16.

[25] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *ACM Symposium on SDN Research*. 164–176.