VTL: A Virtual Transport Layer for Deployable and Configurable Transport Protocols

El-Fadel Bonfoh, Christophe Chassot, Samir Medjiah

VTL: A Virtual Transport Layer for Deployable and Configurable Transport Protocols

https://vtl-project.net

El-Fadel Bonfoh
LAAS-CNRS
University of Toulouse
Toulouse, France
efbonfoh@laas.fr

Christophe Chassot
LAAS-CNRS
University of Toulouse
Toulouse, France
chassot@laas.fr

Samir Medjiah
LAAS-CNRS
University of Toulouse
Toulouse, France
medjiah@laas.fr

***Abstract*—Internet and its evolutions are fundamentally based on the unique TCP/IP model, whose primary protocol of the Transport layer (L4) is the TCP (and somewhat UDP). Despite its well-known limitations, TCP is still widely deployed and used on the Internet. Nearly all the literature's propositions to overcome TCP's limitations are *not deployed* in the mainstream operating systems (OS) and/or face *limited use* by the Internet's applications. This situation leads to the *ossification* or *sclerosis* of the Transport layer that is a significant barrier to the introduction of innovations in this layer of the Internet's TCP/IP architecture. Thus, this paper proposes to address this issue and focuses on two main contributions. First, we design and implement a Virtual Transport Layer (VTL) system able to *dynamically deploy* Transport protocols within the end-systems' OS. Second, to facilitate the use and stimulate the adoption of the proposed architecture (and the L4 protocols it deploys), we provide the approach and mechanisms necessary to allow TCP-based applications to use *transparently* any Transport protocol other than TCP. Experiments show the correctness of VTL and its ability to quickly deploy protocols. Further, we show that the deployed protocols achieve significant performances under VTL.**

***Index Terms*—Communication Protocols, Dynamic Deployment, TCP/IP, eBPF/XDP, Prototype/Testbed Experimentation.**

I. Introduction

With the Internet's growth, several Transport protocols have been proposed to continually improve QoS to satisfy the increasing requirements of applications and adapt to various emerging networks. However, the defacto L4 protocol for end-to-end data transfer between applications within the Internet remains TCP (almost 90% of Internet traffic is based on TCP [1]). Indeed, a review of the literature and an analysis of Internet traffic under analyzers such as Wireshark show that despite their convenient conceptual approach and better performances against TCP/UDP in several cases, any new L4 protocol solutions other than TCP and UDP are *(i) faintly deployed* (in worst cases not deployed) and *(ii)* suffer from *limited adoption* by applications on the Internet. This situation prompts the *sclerosis* or *ossification* of the Transport layer that hampers the introduction of innovations in this Internet's layer.

Therefore, this paper proposes to address Transport layer sclerosis at the end-systems by providing *practical* answers to two major questions: *(1) How to (effectively) deploy a specific protocol mechanism at the end-system*? *(2)* And, assuming the availability of protocols on the end-system, *how to ensure its seamless usage by (legacy and aware)*[1] *applications*? Apart from the extensive evaluations performed during our journey, the main contributions of this paper are the following:

- ***(i)*** We introduce the concept of *dynamic deployment* of protocol grafts. Dynamic grafting of protocols consists of *on-the-fly* integration of protocols within the end-system.
- ***(ii)*** To prevent a limited use of the proposed architecture (and the protocols it deploys), we introduce an approach that permits to replace at runtime TCP by another L4 protocol. We achieve it *transparently* to legacy applications, i.e., there is no need to rewrite the latter applications' code.

We realize the above contributions within VTL (for **V**irtual **T**ransport **L**ayer) system that we fully designed and implemented. VTL follows three main design principles: *(1) the seamless support of legacy applications*, i.e., legacy applications might consume Transport services without the need to rewrite their code; *(2) the separation of protocol from aware-application*, i.e., in line with the service-oriented approach [2], aware-application should request Transport services instead of invoking a specific protocol as it is the case in the standard socket API; and *(3) the protocol modularization*, i.e., the Transport layer data plane is organized in such a way to allow the implementation of reconfigurable protocols whose components might be dynamically instantiated and parameterized.

After this introduction, in Section 2, we provide an insightful analysis of the Transport layer ossification causes and then revisit the limitations of previous works that address this issue. This analysis allows us to derive and motivate the conceptual and technical choices made during VTL design

[1]Throughout this paper, a legacy application designates a TCP-based application that uses the standard socket API to consume Transport services. We will use interchangeably the terms "legacy application" and "TCP application". We define *aware-application* as an application that uses the API provided by VTL to consume Transport services.

and implementation. In Section 3, we first introduce VTL's key concepts. Then, we present the functional architecture of VTL. In Section 4, we evaluate VTL. Apart from showing the correctness of VTL, this evaluation shows the deployment delay and the performances (under VTL) of the deployed protocols by taking as reference TCP performances in the same conditions. Finally, Section 5 and Section 6 conclude the paper with a discussion on the limitations and future work of VTL and a summary of learned lessons.

II. Background

A. Vicious Circle

The deployment and wide adoption of any L4 protocol on the Internet rise up several challenges and require taking into account three main stakeholders: *(1)* the run-time environment of the protocols principally managed by *OS developers*, *(2)* the consumers of the services provided by the protocols, namely *application programmers*, *(3)* the *middleboxes vendors* who maintain the network infrastructure over which data packets handled by the protocols are transmitted.

Deployment Barriers. The OS vendors' historical choice to implement L4 protocols in the kernel-space of OS is not without consequence. To provide support for a new protocol, OS developers must integrate it into the kernel which requires an upgrading of their system. OS upgrade is not only time-consuming and very tedious, but it also poses significant software and hardware compatibility issues. Any modification of the kernel code requires the utmost attention to detail. Those modifications are therefore left to the experienced developers of OS vendors. Even for open systems like Linux, it takes *benevolent dictators* to ensure the stability and reliability of the OS. As a result, OS upgrade frequency is slow and for OS developers, only a high demand from application programmers can motivate the integration of any new protocol solution.

Limited Use Root Cause. Programmers must modify their applications' code to adopt any new protocol solution due to the standard *socket API* caveats [3]. This latter corollary might be a factor of increasing complexity and a potential source of instability since it is necessary to rewrite the application each time a new protocol solution is released and best matches the application's needs. Our analysis is that to prevent the issues associated with frequent modifications, most of the application programmers prefer to rely on standard protocols such as TCP or UDP, which are recognized as stable and available on the mainstream OSes and supported throughout the Internet, rather than using a new protocol whose reliability and acceptability are not guaranteed, even if this latter is more appropriate to meet their applications' requirements.

Other Factors Out of End-systems. While the logic that prevailed at the birth of the Internet is based on the *end-to-end principle* [4] where the network, composed mainly of routers, had no visibility over what happens beyond the L3 level, the massive introduction of middleboxes (such as NATs, or firewalls) has "violated" this principle. Those middleboxes can read and modify packets up to level L4 and reject any unrecognized protocol. To meet certain requirements (e.g., security, fast network troubleshooting, etc.), middleboxes vendors (and somewhat network operators) are often unwilling to configure their devices to whitelist any new protocol until the most popular OSes support this latter protocol.

TABLE I
Selected Transport Layer Protocols and Architectures.

	MPTCP	**QUIC**	**PQUIC**	**UTCP**	**NEAT**
Deployment Space	Kernel	User	User	User	User
Protocol-agnostic API	×	×	×	×	✓
Legacy Appli Support	✓	×	×	×	×

All in all, this global context leads to the well-known vicious circle: application programmers are unwilling to use a new protocol that is unlikely to work end-to-end; OS developers will not implement a new protocol if application programmers do not express a need for it; middleboxes vendors (somewhat network operators), will not add support if the protocol is not in mainstream operating systems; the new protocol will not work end-to-end because of lack of support in middleboxes.

Note 1. Through extensive measurements in [5], Honda et al. come to the conclusion that *"the blame for the slow evolution of protocols (with extensions taking many years to become widely used) should be placed on end-systems"*. Therefore, we argue that a handy approach should focus on the end-systems. We preconize systematic fallback to TCP or UDP to prevent complete failure in case of rejection during middlebox traversal.

B. Related Work

Many research propositions have emerged (see Table I) to enrich the Transport layer with additional services and bring to it more flexibility and extensibility. Those works take two research directions: *(1)* proposing a *single/specific protocol* and *(2)* rethinking the *whole Transport layer* architecture.

1) Single protocol propositions:
Extending standard protocols such as TCP and implementing those extensions as a set of kernel patches is a very common approach to add new services to the Transport layer. The main protocol that follows this approach is MPTCP [6] that adds a multipath capability to the regular TCP. However, OS kernel modification is tricky. To overcome the associated constraints (development difficulty, slow updates, etc.), a second approach consists of implementing part of the protocol in the user-space above UDP. The modular ETP [7] and the emerging QUIC [8] protocols follow such an approach.

Deploying a protocol in the user-space above UDP facilitates the protocol extension and increases its update/upgrade frequency. However, performances often take a hit. To speed up the user-space protocol, a third approach is, instead of using UDP as substrate, to implement the whole protocol in user-space on top of *kernel bypass* tools such as DPDK [9] or NetMap [10]. UTCP [5] is one of such protocols that rely on NetMap to implement a full high-performance user-space Transport protocol. However, despite their efficiency, such an approach still poses some security and efficiency concerns.

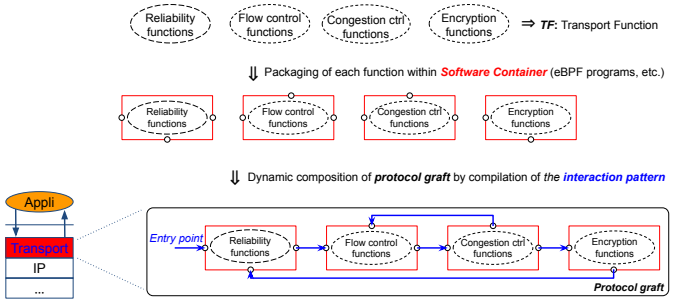

Fig. 1. Conceptual illustrations of Transport function (TF) and protocol graft.

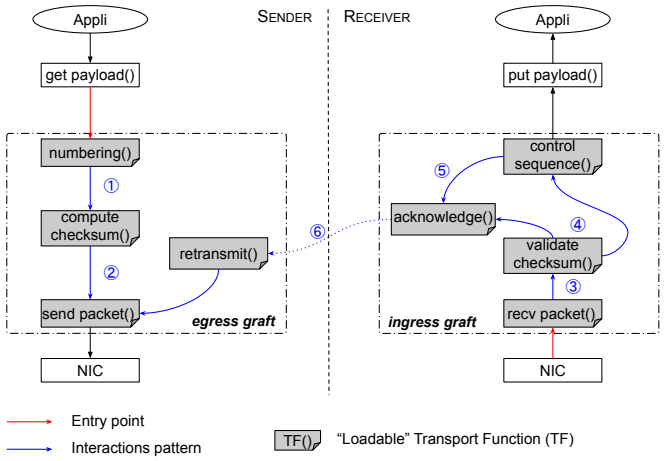

Fig. 2. Illustration of protocol graft providing an ARQ-based reliable service.

Some works in the literature rely on the dynamic deployment of code to extend the Transport layer and make it more flexible. The seminal work that relies on this approach is the STP proposition [11]. STP is a framework that allows the deployment of TCP extensions. Its primary goal is to speed up the upgrading of TCP protocol within the end-system by the use of mobile codes that are exchanged remotely between end-systems. More recently, PQUIC [12] introduced a prototype of a framework able to dynamically extend QUIC protocol by loading at runtime new Transport protocol plugins that contain the code of the mechanisms. PQUIC relies on a *user-space version* of eBPF VM [13].

In summary, each of the above approaches demonstrates how to "force" the deployment of new protocol mechanisms and services within the end-system OS. Besides the specific drawbacks related to each approach discussed above, they have one more additional common limit: they are specific and limited to a single protocol, i.e., they lack a *protocol-independent API* and then force applications to bind to a unique protocol at their design-time and may, therefore, lead to a slow/limited adoption as explained in Section 2.A.

2) Rethinking the whole layer architecture:
Instead of proposing a single protocol, other works propose to rethink the whole architecture of the Transport layer in a way to eliminate the regular socket API limitations. Their main idea consists of replacing the socket API with a common *Transport services interface* (a.k.a. service-oriented API). Contrary to the socket API, the application that uses the service-oriented API will no longer have to choose a unique L4 protocol at its design-time but should instead specify the Transport services it wants. Such an approach aims to ease the adoption of all protocols available on the end-system by *non-legacy* (i.e. aware) applications. In 2014, an IETF working group named TAPS [2] was chartered to promote and lead this approach's standardization efforts. As of the writing of this paper, NEAT [14] is the single official implementation of the TAPS standard.

Despite its potential to simplify the way applications consume Transport services and to break the dependency of the application to a single protocol, the service-oriented approach presents two major drawbacks. First, it relies on the assumption that the Transport protocol to use is already available on the end-system OS. Therefore, it lacks a method to dynamically deploy a new protocol mechanism where appropriate: the protocol's choice depends on the end-system OS network stack. Second, it does not provide any transparent support for *legacy* applications; these applications should be rewritten directly on top of the service-oriented API beforehand to leverage the new architecture. This last limit could be a barrier to the adoption of this approach (see Section 2.A).

III. Virtual Transport Layer System

A. VTL Core Concepts

1) Transport Function (TF):
A *Transport function* (TF) is the most atomic entity of the VTL data plane that executes a single protocol processing logic such as a checksum calculation or packet numbering. It implements a single local function that could roughly be grouped as follows (see Fig. 1): reliability functions, flow control functions, etc. Conceptually, the group to which a TF belongs indicates the Transport service (e.g. a congestion control service) the TF participates in implementing. Indeed, a TF as alone cannot provide any Transport service; it must be composed with other TFs. Each TF must be made *pluggable* thanks to its wrapping in a software container (e.g., eBPF programs, in the current implementation of VTL). The packaging in a software component of a TF will provide it with two essential interfaces: *(1)* the *input interfaces* from which it should receive any data packet or control information; *(2)* the *output interfaces* to which it should push any data packet.

2) Protocol Graft:
As illustrated in Fig. 1, a *protocol graft* is a list of pluggable TFs with their *interaction pattern* that defines the way the TFs are connected, i.e., the sequence in which data are processed by the different TF composing the graft. A protocol graft might provide one or more *services* such as reliability service, ordering service, and so on. Contrary to a TF which is located on a single side of the Transport session, a protocol graft is conceptually distributed between the sender end-system and receiver one and provides comprehensive *Transport service(s)/feature(s)*. The sender side maintains the *egress graft*, whereas the receiver side maintains the *ingress graft*. Finally,

at each side of a session, a protocol graft should create and maintain a memory buffer shared between TFs serving to ensure data persistence. Fig. 2 illustrates a protocol graft distribution for a typical ARQ-based reliable service; note that this is only for illustration purposes; a more comprehensive reliable service should have many other functions such as duplication control.

3) VTL Services and Features:

VTL core functionality is to deploy the appropriate protocol graft on behalf of the application in order to ensure that data are moved according to the application requirements. From an internal point of view, VTL: *(i)* ensures the Transport session establishment following a classical client (connect) / server (accept) model; *(ii)* ensures the selection, negotiation, and deployment of the protocol graft; and *(iii)* monitors both the network and application states in order to adapt the protocol graft at runtime to the change of environment. VTL provides by default a message-stream-oriented service that should be replaced if the underlying deployed protocol graft provides a byte-stream-oriented one. VTL expects to provide protocol developers with the facilities to write, test, and publish new protocols within the end-systems.

VTL allows applications to express their requirements by firstly characterizing the desired Transport services and then associating those Transport services with a set of desirable QoS parameters. For instance, a requested Transport feature/service might be expressed in terms of *reliability*, *order*, *congestion control*, *security*, etc. In addition to these Transport features, associated QoS parameters can be expressed in terms of maximum acceptable *delay* (ms), minimum *throughput* (Mb/s), and allowable loss rate (percentage).

B. VTL Architecture

VTL components (see Fig. 3) are separated between two planes: a *control plane* and a *data plane* constituted mainly by the runtime environment of Transport functions (TFs), i.e. the eBPF VM. Both planes share information (ACK, KTF negotiation state, packets, ...) thanks to a set of shared *MAPS*. VTL defines two workflows: *KTF deployment workflow* and *data delivery workflow* that we further described latter.

Control Broker is responsible for ensuring the protocol graft negotiation at the end of which the appropriate TFs should be deployed at the sender side as well as the receiver side of the Transport session. *Launcher* component, driven by *Control Broker*, is in charge of the configuration and the instantiation of the requested TFs inside the *eBPF Kernel VM*. To observe the network and measure its key quality parameters (delay, loss rate, and throughput), VTL provides *NetMonitor* component. This component captures the network parameters mentioned earlier and reports them to the *Control Broker* component *on-request* for further analysis.

VTL socket is a data plane structure manipulated through the *protocol-agnostic API* by aware-applications to send and receive data. It is a *virtual socket* that emulates either a RAW socket for data transmission and/or an XSK socket for data receipt. VTL socket is created, maintained, and exposed to applications by *Data Broker* that ensures data moving and dispatching between the application buffers and the deployed TFs running inside the *eBPF VM* component.

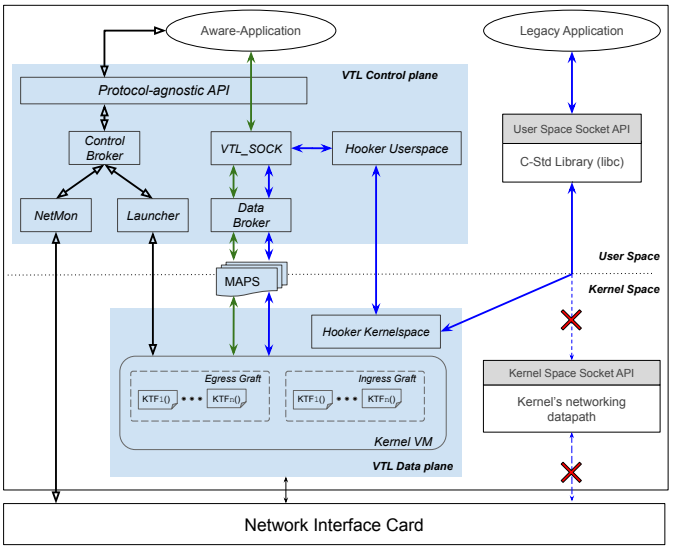

Fig. 3. VTL system architecture overview. Blue arrows represent the legacy application datapath, green arrows the aware-application datapath, and black arrows the KTFs deployment path.

C. VTL Implementation

1) Background:

VTL implementation relies on the eBPF [13], recently introduced in the Linux OS. eBPF is an extended version of BPF [15] and it allows injecting bytecode at runtime within the OS kernel. Its usage scenarios cover filtering, networking, systems' security, etc. eBPF infrastructure is constructed around three major elements: *maps, tail calls,* and *helper functions*.

Maps are data structures storing a set of {*key, value*} pairs used to exchange data either between user-space programs and in-kernel eBPF programs or between eBPF programs running at different points of the kernel. Maps, often attached/pinned to the root file system (i.e. `/sys`), are also useful to ensure data persistence between successive invocations of eBPF programs. In its early versions, eBPF limits each program to a maximum size of 4096 BPF instructions. In order to overcome this size limitation, eBPF integrates the concept of *tail calls* that could be used to chain up to 32 different eBPF programs; tail calls feature is an enabler of modularization's implementation. Nevertheless, it is worth noting that since Linux version 5.2.0, released in 2020, an eBPF program can contain up to 1M (one million) instructions. Basically, *helper functions* define a list of functions that an eBPF program can call during its execution. Thanks to helper functions (and eBPF verifier), access to kernel by eBPF programs is strictly controlled to prevent OS damage. In other words, helper functions allow eBPF programs to interact directly with the kernel safely.

Each eBPF program that is deployed inside the OS kernel has a specific type and must be attached to a *hook point*, also known as a *kernel event* (incoming packet, system calls, socket

TABLE II
MAIN ADDED HELPER FUNCTIONS WITHIN THE VTL SYSTEM.

Helper functions	*Description*
`vtl_start_timer(i, n)`	Set timer *i* to n ms
`vtl_stop_timer(i)`	Stop the timer *i*
`vtl_build_graft()`	An exogenous wrapper of `tail_call()`

operations, etc.). Then, each time the event occurs, the eBPF program attached to it is executed.

Note 2. To deploy Transport functions in the OS, we first considered the use of LKM approach [16]. As eBPF, LKM allows *hot* plugging of modules in the OS kernel thanks to tools such as `modprobe`. Unfortunately, during our first prototyping, we face the most common issue of kernel module utilization: the whole system frequently crashes at the slightest mistake. Indeed, there is no verifier to guarantee the safety of the OS. This is a major difference with eBPF programs; kernel modules are entirely part of the kernel. As such, they have the same rights as the kernel itself and can perform without any control, any operations (call any kernel functions, access any memory buffer and register, etc.). Furthermore, there is a lack of kernel debug tools and it takes time to troubleshoot the code and repair the bug.

2) Implementation Overview:
VTL combines two kernel subsystems: `XDP` and `TC` [17], part of eBPF. This association permits *(i)* to grasp the outgoing packets as late as possible just before they reach the network interface card (NIC) and *(ii)* to pick up the incoming packets as early as possible before they reach the OS network stack.

KTF: an eBPF instantiation of TF. In the current implementation of VTL, a Transport function (TF) is instantiated in the form of eBPF programs and called KTF for *Kernel Function Transport*. KTF inherits common properties of an eBPF program, i.e.: ***(1) Input interface***: a *hook point* serving as an entry point of the function (`TC` to get all egress packets, and `XDP` to pick all ingress packets); ***(2) Shared Memory***: *MAPS*, serving as *data buffers* to store packets for eventual retransmission, or to share control information with the user-space programs such as a list of already acknowledged packet or KTFs negotiation state; ***(3) Output interfaces***: *helper functions* used to implement the core algorithms of the protocols and serving as output points towards the next KTF, the application, or the network.

VTL extends helper functions provided by the native eBPF VM in order to address some specificity related to L4 protocols functioning. For instance, a "simple" stop-and-wait protocol needs a timer to trigger packet retransmission in case of loss. Nevertheless, there is currently no timing control helpers. VTL then adds and exposes to KTFs a set of helper functions necessary to start and stop a timer. Table II summarizes the set of helper functions added by VTL.

3) Aware-application Session Initiation:
Obtaining VTL Socket. Prior to all, the aware-application must obtain a VTL socket. The application indicates its transfer mode in one of the following modes: sender, receiver, or both. Based on the transfer mode specified by the application, *Control Broker* configures a new VTL socket and its associated buffers (see Fig. 4) and triggers the deployment of the ***canonical graft*** (see Section 4). Finally, *Control Broker* associates the deployed canonical graft's file descriptor to the VTL socket and gets back the resulting VTL socket structure to the application. At this stage, the application gets a ready VTL socket that it could use to send and receive its data.

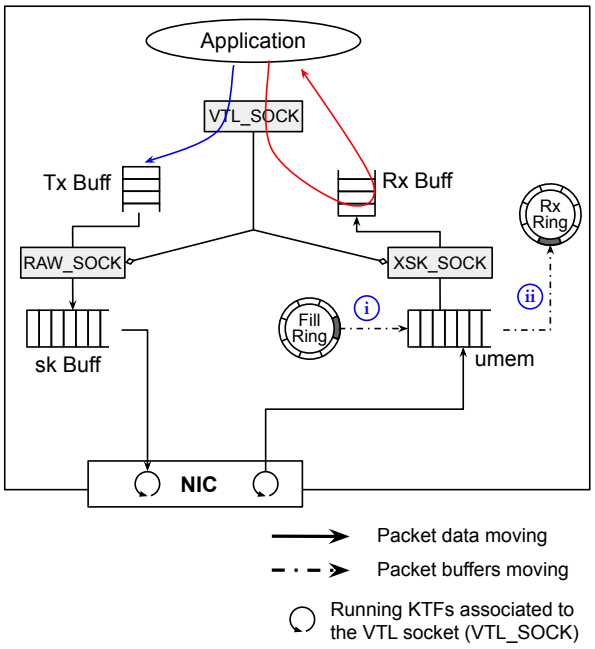

Fig. 4. Data moving between application and VTL socket and its buffers.

Protocol Grafts Negotiation and Deployment. Protocol grafts negotiation process between a sender and receiver is shown in Fig. 5. Each side of the connection maintains its own map named `qos_nego_MAP`. Each index of the `qos_nego_MAP` associates the file descriptor value of the VTL socket and the associated graft negotiation outcome: `N_ACCEPT` or `N_REFUSE`. At the sender side, the canonical graft named *egress_cano_graft* runs two KTFs: one `TC` section named `egress_tf_sec` and one `XDP` section named `listener_tf_sec`. The receiver side canonical graft, named *ingress_cano_graft*, executes a single `XDP` program section named `ingress_tf_sec`.

Sender: the client of the negotiation. Aware-application that has specific requirements defines them by invoking the *protocol-agnostic API* ①. Based on predefined matching rules, *Control Broker* selects in the *KTFs pool* the most appropriate egress and ingress grafts to meet application needs. Then, it pre-builds a negotiation packet and transmits it to the IP layer ②. It then waits for a while before looking up the negotiation state in the `qos_nego_MAP` which must be updated by `listener_tf_sec` at the receipt of the receiver reply. The packet is finally transmitted to the network device ③. The negotiation acknowledgment packet, sent by the receiver ⑦.② in reply to the negotiation request, is intercepted and processed

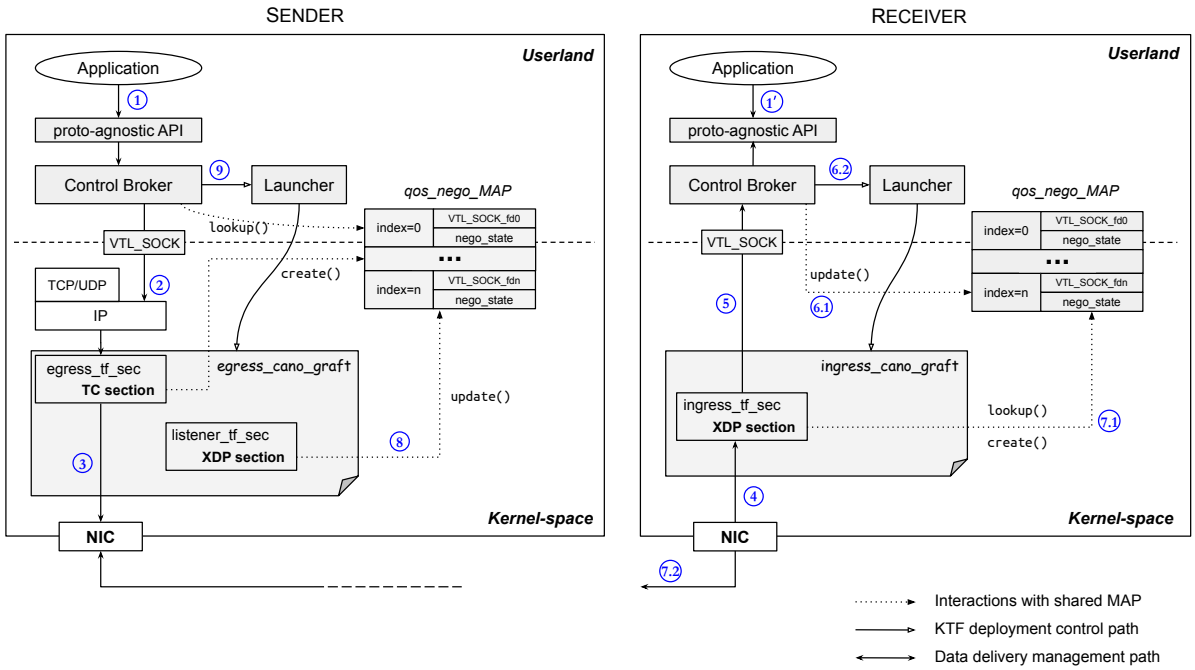

Fig. 5. Protocol grafts negotiation process under VTL system.

by `listener_tf_sec` which updates the `qos_nego_MAP` to signal to *Control Broker* the receiver response (8). In the case of acceptance (value set to `N_ACCEPT` in the map), *Control Broker* triggers the deployment of the selected egress graft to replace the canonical one (9).

Receiver: the server of the negotiation. After the application has finished obtaining a VTL socket, it issues a blocking request (1') to expect and retrieve the outcome of the negotiation handled by *Control Broker*. At the receipt of a *negotiation* packet from the sender (4), `ingress_tf_sec` delivers it to *Control Broker* (5). Then, it consults the *KTFs pool* to confirm the availability of the ingress graft requested by the sender and updates the `qos_nego_MAP` to `N_ACCEPT`; if the availability of the requested KTF is not confirmed, the map is updated at `N_REFUSE` (6.1). In the case of availability of the requested graft, after updating the map, *Control Broker* triggers the requested ingress graft deployment (6.2). Each time it receives and passes a negotiation packet, `ingress_tf_sec` waits for a while, then reads `qos_nego_MAP` to check the decision taken by *Control Broker* (7.1) and ends up by sending acknowledgment packet to the sender (7.2). This acknowledgment packet should take one of the two following values: `NEGO_ACK` in case *Control Broker* validates the graft negotiation request or `NEGO_NACK` if not.

4) KTFs Deployment Workflow:
When *Control Broker* requests the deployment of a specific KTF stored as an object file, *Launcher* picks it from the *KTFs pool* and starts its loading. *KTFs pool* is a repository of a set of precompiled and ready to be deployed KTFs. The precompilation of KTF in the form of an object file eliminates overheads of the Clang/LLVM compiler during the deployment of TFs (see Section 4). Before its effective loading, a KTF is checked by the verifier via a series of *verifications*[2] to ensure that the deployed KTF will not crash the OS. Once the verifier finishes its checking, the KTF is compiled by the JIT compiler in the eBPF native assembly code of the end-system CPU. The loaded KTF is finally attached to the network interface and ready to process all incoming packets if attached to the `XDP` hook and all outgoing packets if attached to the `TC` hook.

5) Data Delivery Path:
Fig. 4 depicts how internally the VTL socket transfers data through its associated buffers. During data moving, the interactions between the application and VTL system are performed asynchronously thanks to pairs of buffers at the transmission (*Tx buff* and *skb buff*) and at the reception (*Rx buff* and *umem*). Application ready to send data puts it in its *Tx buff* where *Data Broker* picks it up, forms VTL packet payload, and pushes it on the *skb buff* for the IP layer.

At the reception, the received data might be sent directly to *Data Broker* in userland for fast delivery to the application. With the aim of making use of XSK socket zero-copy capability, *Data Broker* and the OS kernel share the *umem* buffer. Since the *umem* buffer is shared, the memory access conflicts and deadlock events might happen. To prevent that, VTL leverages AF_XDP socket family [18] features to associate two ring buffers to the *umem* buffer: the *fill ring*

[2]A common list of verifications include but are not limited to: *(i)* the syntax of C code instructions, is there any infinite loop without explicit stop condition; *(ii)* the way the protocol component interacts with the kernel, is there any use of unknown or unauthorized helper function; *(iii)* the memory access, does the KTF try to access a specific memory without prior check the accessibility of this memory; *(iv)* the number of instructions in the KTF that must be under 1M (4096 for all Linux version prior to v5.2.0).

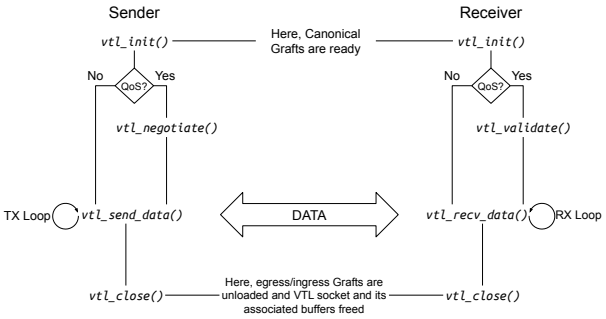

Fig. 6. Typical function call flow by VTL aware-applications for data Tx/Rx.

and the *Rx ring*. The former is used by *Data Broker* to pass the ownership of the packet buffer to the kernel (ⓘ in Fig. 4) whereas the kernel uses the *Rx ring* to pass the ownership of packet buffer to *Data Broker* (ⓘⓘ in Fig. 4). In this way, when the kernel receives a buffer on its *fill ring*, it knows that the *umem* memory space associated with the buffer is free and that it can safely put incoming frame data on this space. In the same way, when *Data Broker* receives a buffer on its *Rx ring* it knows that the *umem* space associated with that buffer is free and that it can pick up the data there without conflict with the kernel. Finally, *Data Broker* makes payload data available on the *Rx buff* for the application where this latter may retrieve it by making use of the *protocol-agnostic API*.

6) A Typical Transport Session of Aware-application: *Protocol-agnostic API* component ensures the principle of the separation between application and protocol. It is a shared library used by applications to express their requirements and to send/receive data. The *protocol-agnostic API* is easily extensible and may integrate other functions in the future to respond to more extensive use cases. We illustrate the currently implemented functions through a function call flow in Fig. 6. Table III summarizes the way applications should specify parameters when using the *protocol-agnostic API*.

7) Hooker: Legacy Applications Integration: *Hooker* component's goal is to provide support to legacy applications. The internal structure of *Hooker* is shown in Fig. 7. *Hooker* attaches to the root `cgroupv2` [19]; therefore, by making use of the hierarchical model of cgroups, it processes every ingress and egress packets of all processes running on the end-system. *Hooker* maintains a map of type `SOCKMAP` that key is a structure containing the addressing information. This key is used by the *msg_redirector* program to identify the right socket towards which the packet data must be forwarded to. Each time a connection is established or closed by one process, the map is updated by *msg_redirector* thanks to a `SOCKS_OPS` bpf program section attached to `cgroupv2`.

Every time an application process sends a data packet by calling into `sendmsg()` on a TCP socket, the `SK_MSG` bpf program running by *msg_redirector* intercepts it, rewrites it if necessary thanks to the helper function `bpf_msg_push_data()`. Finally, to deliver the message either to the redirection socket or

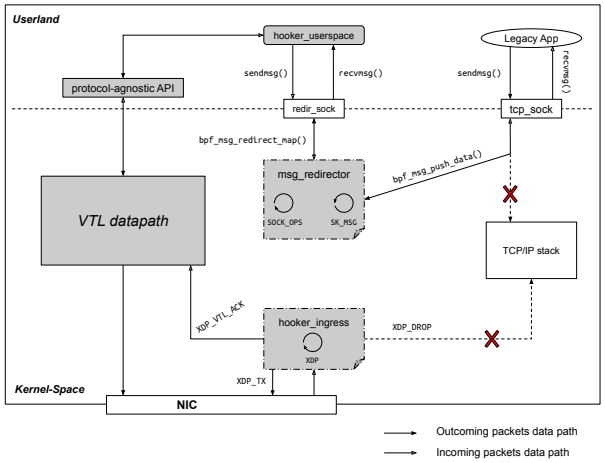

Fig. 7. VTL *Hooker* Component Internal Structure.

to the TCP socket, *msg_redirector* program leverages `bpf_msg_redirect_map()` helper function. The redirection socket is created and maintained by *Hooker userspace* program which will use the `recvmsg()` operation to get the redirected data packet and send it to the VTL datapath through the *protocol-agnostic API*. At the reception, once the host network interface (NIC) receives data packet, the `XDP` program running by *Hooker ingress* intercepts the data packet and processes it by issuing the right verdict. The *Hooker ingress* program can drop the packet data (`XDP_DROP`), redirect it to the same NIC (`XDP_TX`) or, as currently done, pass it to the ingress VTL datapath (`XDP_VTL_ACK`) for further processing.

IV. Experiments and VTL Evaluations

A. Implemented KTFs and Grafts

We revisited and implemented, *from scratch*, a set of protocol mechanisms well-known in the literature namely and ARQ reliable graft based on Go-back-N, a Selective Repeat (SR) graft, and a Partial Reliable (PR) graft. Code Listing 1 illustrates a *template* of an egress graft part of a protocol.

Canonical Grafts purposes are *(i)* to enable the immediate transfer of data of applications that do not have specific requirements and *(ii)* to conduct the KTFs negotiation stage for QoS-oriented applications. Canonical grafts are deployed at the creation of a new VTL socket to which the KTFs composing the canonical grafts are associated by default. For each packet it processes, the *egress* canonical graft sets up the *type* header field of the packet (either to `DATA` or to `NEGO`), ensures the processing of acknowledgment packet and signals to *Control Broker* the receiver's reply thanks to the shared `MAPS`. On its side, at each packet it receives, the *ingress* canonical graft extracts the *type* value of the packet before passing it either to *Control Broker* or to *Data Broker* in userland. When the packet type is `DATA`, the ingress canonical graft immediately passes it to the *Data Broker* and continues processing the next incoming packet. Otherwise (i.e., the received packet is negotiation one), it waits

TABLE III
VTL PROTOCOL-AGNOSTIC API FUNCTIONS PARAMETERS.

Functions	Parameters	Description
`vtl_init()`	`mode, src_ip, dst_ip`	Create a new VTL socket and its associated resources (buffers, canonical grafts, etc.).
`vtl_negotiate()`	`vtl_sock,l4_services, qos_values`	Get application's requirements and trigger a graft negotiation process.
`vtl_validate()`	`vtl_sock`	Retrieve graft negotiation.
`vtl_send_data()`	`vtl_sock, buffer, buffer_size`	Send payload and retrieve the size of written data.
`vtl_recv_data()`	`vtl_sock, buffer, buffer_size`	Fetch payload and return the read data size.
`vtl_close()`	`vtl_sock`	Close VTL socket and free resources (buffers, KTFs, etc.)

for the outcome of negotiation handled by *Control Broker*, transmits an acknowledgment to the sender, and pursues the next packet's processing.

Partial Reliable (PR) Graft. Partial reliability concept consists of allowing KTFs not to issue at reception all the data packets submitted by the sender, provided to respect a maximum percentage *MAX_LOSS* of allowable losses (e.g. *20%* of the packet data may be lost). The goal is to deliver, as quickly as possible, the out-of-sequence packets data to applications that tolerate a certain amount of loss.

```
#include <vtl.h>
// and other useful headers

/* Declare a MAP to store data packet for retx */
struct bpf_elf_map SEC("maps")
EGRESS_PKT_WND_MAP = {
 .type=BPF_MAP_TYPE_HASH,
 .size_key=sizeof(int),
 .size_value=sizeof(vtl_pkt_t),
 .pinning=PIN_GLOBAL_NS,
 .max_elem=16,
};

SEC("egress_tf_sec")
int _tf_tc_egress(struct __sk_buff *skb) {
// skb is the entry point of the TF

/*** TF code here ***/
}

SEC("listener_tf_sec")
int _listener_tf(struct xdp_md *xskb) {
// xskb is the entry point of the TF

/*** TF code here ***/
}
```

Code listing 1: Template of protocol graft and Kernel Transport Function (KTF).

B. Testbed Setup and Methodology

We implemented and evaluated VTL under Linux 5.3.5. VTL experiments are performed under a testbed constituted by two hosts linked by one router. The router and the associated network parameters are emulated thanks to `netem` tool [20]. Unless otherwise stated, the network parameters used during experimentation are the following: RTT=10 ms, bandwidth=16 Mbps and loss rate $\in$ [0, 5%]. Each host is equipped with Intel Core i7-7500U CPUs, 3.8 GiB RAM, and Qualcomm Atheros QCA6174 NIC driver.

Further, we implemented two VTL aware-applications, one acting as a *data* streaming server and the other one playing the client role. The server application is able to stream several kinds of files with different sizes and formats. The reference we used to discuss the obtained results is TCP (cubic) that we evaluated in similar network conditions. The window size of the Go-Back-N, the Selective Repeat, and the partial reliability grafts is set to 16. Finally, to avoid interference with packets not directed to VTL within the testbed environment, we set the IP protocol number of VTL packets to experimental hexadecimal value `0xfd` [21].

C. Microbenchmarks

1) KTFs Size and Graft Negotiation Delay:
First, we assessed the *delay required to complete protocol graft negotiation procedure*. This delay is composed of *1/* packets processing delay (negotiation one and its associated acknowledgement), and *2/ KTFs deployment delay* at both sides (sender and receiver). Nevertheless, the carried-out experiments demonstrated that the packet processing delay is negligible compared to the delay of KTFs deployment precisely when the RTT is low (as that is the case here). Consequently, the results reported in Fig. 8 illustrate essentially the delay of KTFs deployment within the end-system OS. We compute delays with the help of `time` tool [22].

The delay of *one* KTF deployment is the sum of the *compilation delay* (T0) and the *loading delay* (T1+T2). The former measures the time required by the userland compiler Clang/LLVM to compile the KTF from a source file to object file. Further, the loading delay is the total time elapsed during verifier/JIT operations (see footnote 2 in Section 3) and the `XDP` (resp. `TC`) hooks attaching performed by `iproute2` [23] (resp. `tc` [17]) tool. Based on this breakdown of the negotiation delay, to reduce the deployment delay (i.e. the negotiation delay), the first and intuitive approach is to pre-

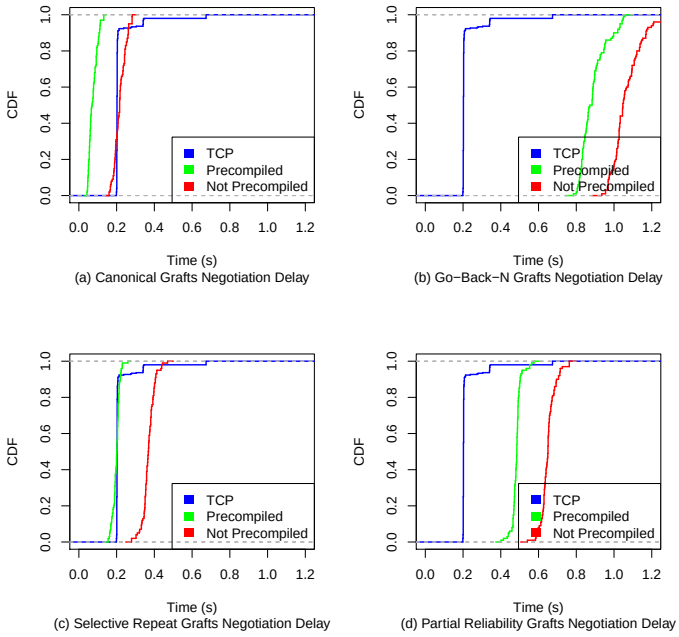

Fig. 8. Negotiation delay. Precompiled grafts (green) considerably reduce negotiation delay. As a reference, the curve in blue illustrates the latency of TCP's 3 way-handshake connection setup with RTT at 100 ms and loss rate at 2%.

compile and to store KTFs/grafts as an object file rather than a source file. This will obviously eliminate userland compilation delay during the deployment. The results reported in Fig. 8 validated this intuition and most importantly showed that the *benefit of precompiled grafts is not negligible*. In fact, we found that when the KTFs are precompiled and stored as object files, the total delay of protocol grafts negotiation could be reduced by 20% to 60%. Moreover, we noted that (excepted Go-Back-N) the *negotiation delay increases with the size of the grafts*, which makes sense in view of the important part of the deployment delay over the total cost of the negotiation procedure.

Given the significant reduction enabled by precompiled grafts on the negotiation delay, one might be tempted to precompile all grafts and store them as object files in the *KTFs pool*. Nevertheless, a closer look at Table IV that provides the statistics on the complexity of the grafts shows that even if the precompiled grafts have a definite advantage on the negotiation delay, they are bulkier than the *non*-precompiled grafts (i.e. stored as source files). For example, for a canonical graft, it takes 6 times more memory space to store its precompiled version (33.1 KB) than its source version (5.58 KB). If the *KTFs pool* is small and the end-system has a large storage capacity (as in most commodity computers.), the size of the precompiled grafts will not be a limit. This will not be the case when the *KTFs pool* will store more and more protocol grafts or when the end-system will have less storage capacity (such as on a microcontroller). Moreover, in a scenario where protocol grafts, instead of being stored locally, should be retrieved from a remote server, a large graft will undoubtedly

TABLE IV
THE CODE COMPLEXITY OF THE IMPLEMENTED GRAFTS.

Grafts	SLoC			Source File Size			Object File Size		
	egress	ingress	Total	egress	ingress	Total	egress	ingress	Total
Canonical	102	65	**167**	3.45 KB	2.13 KB	**5.58 KB**	21.4 KB	11.7 KB	**33.1 KB**
Go-Back-N	187	81	**268**	8.12 KB	2.84 KB	**10.96 KB**	25.6 KB	14.7 KB	**40.3 KB**
Selective Repeat	193	65	**258**	8.83 KB	2.21 KB	**11.04 KB**	26.8 KB	13.7 KB	**40.5 KB**
Partial Reliability	206	104	**310**	9.13 KB	3.07 KB	**12.2 KB**	27.8 KB	14.5 KB	**42.3 KB**

take longer to be downloaded. This will have a negative impact on the delay of the graft negotiation phase. Finally, the impact of protocol graft negotiation delay should be more or less cushioned by the actual duration of the transfer of the application's data. That is to say, for a short duration data transfer, it might be more interesting to keep the canonical grafts whereas for a transfer of large amounts of data, the application could afford to trigger and wait for the completion of a negotiation procedure more.

2) Data Moving Performances: latency & throughput: Second, we evaluated each implemented protocol graft's performances within several network conditions by the variation of the network loss rate. The results are reported in Fig. 9 and show the file completion time (**FCT**) and the data transfer speed rate. We define FCT as the amount of time elapsed between the first packet sent and the last packet received during the Transport session. The acceptable loss rate in partial reliability graft is set approximately at 10%. According to [7], this is the maximum loss rate that can be resorbed by adaptive coding for some applications such as multimedia transfer.

When there is no packet loss in the network, all protocol grafts have almost equal performances. For small files ($\leq$ 8 MB), TCP clearly presents the lowest performances. However, when the file is a bit larger ($>$ 8 MB), TCP achieves better throughput and FCT than all protocol grafts executed in VTL. Under packet losses, as expected, Go-Back-N graft presents the worst performance. The severity of this poor performance is proportional to the level of the loss rate. Contrary to the lossless network environment where TCP always presents better performance than all protocol grafts when the file size is large, we could note in Fig. 9 (e) and (f) that partial reliability graft has better performance than TCP. Two factors could explain this: first, the partial reliability graft does not systematically retransmit all windows containing lost packets unless the number of lost packets is greater than the acceptable loss rate (2 out of 16 packets $\simeq$ 10%). Second, it is well-known that when TCP experiences losses, it exponentially decreases its congestion window (i.e. its data sending speed rate).

Finally, we demonstrated that VTL can perform a runtime reconfiguration of protocols based on a straightforward predefined reconfiguration rule. The metric under monitoring by *NetMonitor* component during this test scenario is the RTT that is reported with 0.5 ms periodicity. That is to say, the RTT of the network link is calculated every 0.5 ms.

The rule is the following one: when $RTT < 300ms$, VTL deploys and maintains the use of the Go-Back-N protocol graft and as soon as $RTT \geq 300ms$, it systematically triggers the replacement of Go-Back-N by the partial reliability protocol graft with *MAX_LOSS* keep at approximately 10%.

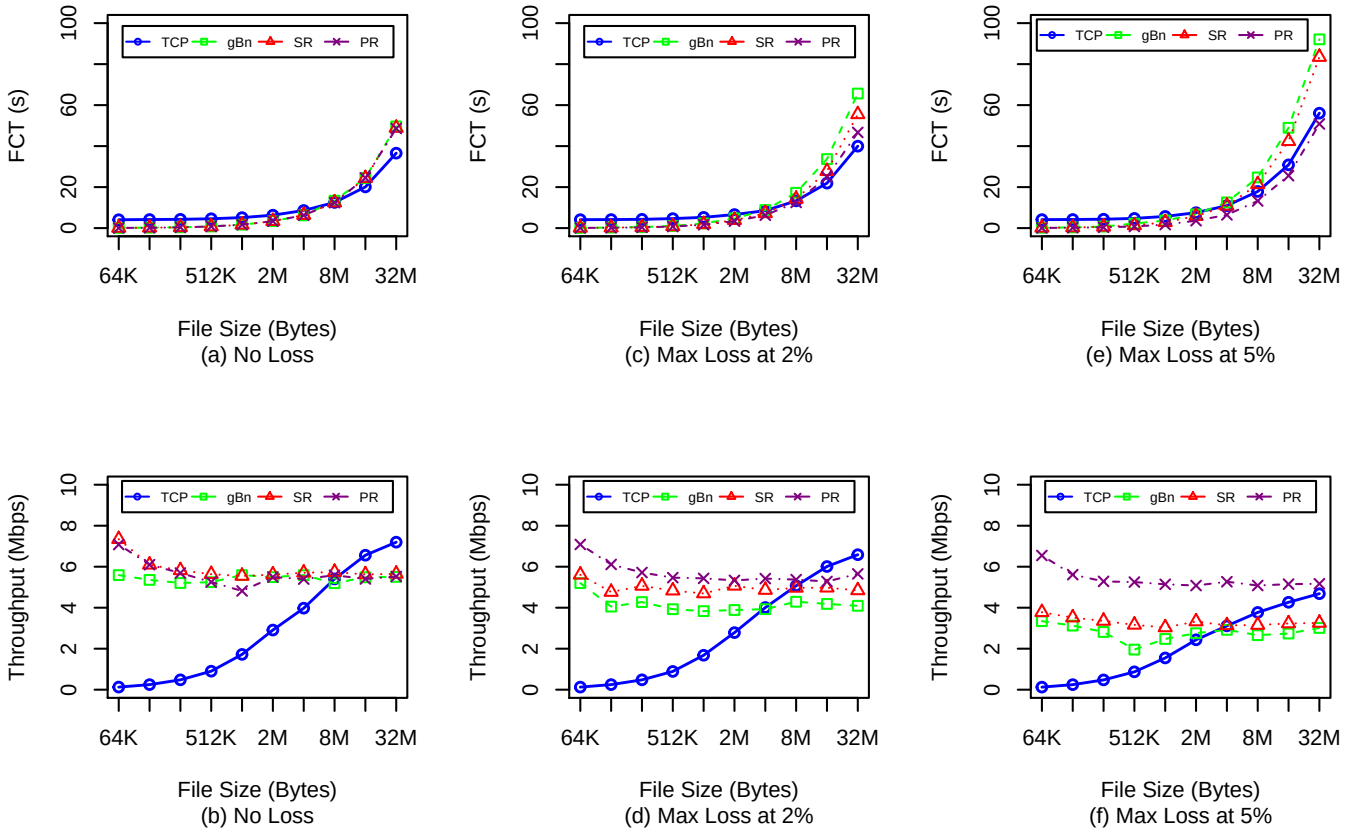

Fig. 9. Data moving performances under various network conditions of protocol grafts: Go-Back-N (gBn), Selective Repeat (SR) and Partial Reliability (PR).

The file used in this use case is a 32 MB video file. We evaluated two types of applications: VTL aware-application and TCP application. For each application, the scenario is identical. During the first 20 seconds, data packets are transferred under RTT of 20 ms. Between the 20th and the 25th second, we switched the RTT from 20 to 320 ms and observed the adaptation actions implemented by the VTL. The results (see Fig. 10) show that VTL is able to adapt to the network conditions in order to limit performance degradation. Indeed, we can observe in Fig. 10 that when an additional delay is introduced at ˜22 ms, TCP throughput significantly decreases. This decrease remains permanent. However, VTL first experiences the effects of the increased delay before slightly offsetting the impacts of the change in network context by replacing the Go-Back-N protocol graft with partial reliability one.

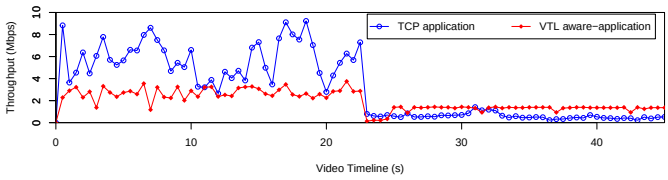

Fig. 10. Throughput variation of VTL aware-application in changing network conditions.

V. Limitations and Future Work

The protocols we have relied on to assess and demonstrate VTL's capabilities are basic mechanisms. A quite feasible and relevant perspective would be implementing more complex mechanisms such as those of the QUIC or MPTCP protocols in the form of eBPF programs. This enrichment of the protocol mechanisms set would lead to the extension of the carried out evaluations and to the reinforcement of the results obtained.

Although conceptually robust, eBPF technology presents some limitations related to the current implementation choices of some of its components, notably `SOCKMAP`. In the implementation of the *Hooker* component of VTL, these limitations have led us to not being able to bypass TCP socket calls without "going back" from the kernel-space to the user-space. At the cost of an implementation effort (and potentially higher complexity), we could initially consider replacing `SOCKMAP` with a `DATAMAP` which would allow the *Hooker* to share data with the application directly in the kernel without the need to open and manage additional sockets from the user-space. A contribution to the eBPF community (more broadly to Linux's one) to address this limitation is a possible technical area of future work.

VI. Conclusion

In this paper, we introduced the VTL system that can *(i)* timely and effectively deploy Transport protocols within the

OS kernel and *(ii)* ensures the flexible use of the deployed protocol by any application, i.e., aware and legacy ones. Following a modular approach, we implemented each considered Transport protocol component as a set of basic *Transport functions* called TF. In the current implementation of VTL, each TF is implemented in the form of an eBPF program that can be deployed at runtime in the kernel-space of the operating system. The deployed TF in kernel-space is called KTF (that is a pluggable form of TF) and could be composed with other KTFs to form a comprehensive protocol mechanism that we called a protocol graft. To evaluate the dynamic deployment capability of VTL, we implemented *from scratch* a set of protocol grafts. During our experiments, we found that VTL could speedily deploy the protocol grafts and KTFs, notably when the deployed grafts and KTFs are precompiled and stored in a dedicated repository. Finally, we evaluated the implemented protocol mechanisms' performances and showed that they achieve excellent performances under VTL. The reference during this evaluation was TCP (cubic) performances in the same testbed configurations.

REFERENCES

[1] D. Murray *et al.*, "An analysis of changing enterprise network traffic characteristics," in *2017 23rd APCC*. IEEE, 2017, pp. 1–6.

[2] T. Pauly *et al.*, "An architecture for transport services," *Internet-Draft draft-ietf-taps-arch-00, IETF*, 2018.

[3] D. Coffield and D. Shepherd, "Tutorial guide to unix sockets for network communications," *Computer Communications*, 1987.

[4] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 277–288, 1984.

[5] M. Honda *et al.*, "Rekindling network protocol innovation with user-level stacks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 52–58, 2014.

[6] A. Ford *et al.*, "Tcp extensions for multipath operation with multiple addresses," RFC 6824, Tech. Rep., 2013.

[7] N. Van Wambeke *et al.*, "Atp: A microprotocol approach to autonomic communication," *IEEE Transactions on Computers*, vol. 62, no. 11, pp. 2131–2140, 2012.

[8] A. Langley *et al.*, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the SIGCOMM Conference*, 2017, pp. 183–196.

[9] L. Foundation, "Myth-busting dpdk in 2020. revealed: the past, present, and future of the most popular data plane development kit in the world," White Paper, Tech. Rep., 2020.

[10] L. Rizzo, "Netmap: a novel framework for fast packet i/o," in *21st USENIX Security Symposium*, 2012, pp. 101–112.

[11] P. Patel *et al.*, "Upgrading transport protocols using untrusted mobile code," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 1–14.

[12] Q. De Coninck *et al.*, "Pluginizing quic," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 59–74.

[13] M. Fleming, "A thorough introduction to ebpf," *Linux Weekly News*.

[14] N. Khademi *et al.*, "Neat: a platform-and protocol-independent internet transport api," *IEEE Communications Magazine*, vol. 55, no. 6, pp. 46–54, 2017.

[15] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture." in *USENIX winter*, vol. 46, 1993.

[16] W. Mauerer, *Professional Linux kernel architecture*, 2010.

[17] B. Hubert *et al.*, "Linux advanced routing & traffic control howto," *Netherlabs BV*, vol. 1, 2002.

[18] M. Karlsson and B. Töpel, "The path to dpdk speeds for af xdp," in *Linux Plumbers Conference*, 2018.

[19] T. Heo, "Control group v2. oct. 2015. url: https://www. kernel. org/doc," *Documentation/cgroup-v2. txt*.

[20] S. Hemminger *et al.*, "Network emulation with netem," in *Linux conf au*, 2005, pp. 18–23.

[21] J. Reynolds and J. Postel, "Rfc1700: Assigned numbers," 1994.

[22] L. manuel page, "time - time a simple command or give resource usage," https://bit.ly/3nStJub, accessed 2020-11-25.

[23] A. Kuznetsov, "Iproute2 utility suite howto," 1998.