# VTL: Timely Deployment and Seamless Adoption of Network Protocols

El-Fadel Bonfoh, Samir Medjiah, Djolo Cédric Tape, Christophe Chassot

# VTL: Timely Deployment and Seamless Adoption of Network Protocols

El-Fadel Bonfoh, Samir Medjiah, Djolo Cédric Tape, and Christophe Chassot

***Abstract*—Internet Transport layer plays a major role in the end-to-end QoS delivered to applications on end-systems. Until now, it has been subject to several propositions to improve its extensibility and adaptability to the rapid evolution of the Internet. Unfortunately, most of these proposals face deployment obstacles and encounter shy-of-adoption on the Internet. This leads to sclerosis of the Transport layer. In order to address this issue, we propose VTL, a protocol deployment and data delivery management system aiming to *timely* and *dynamically deploy protocol* mechanisms to ensure optimal data moving between end-systems. VTL provides *support for legacy applications* without requiring their modification. VTL pushes innovation at the Transport layer by enabling more agility and flexibility in the provisioning of Transport services to applications. VTL is built on a combination of two Linux kernel subsystems: `TC` that hooks under IP layer to process outcoming packets, and `XDP` that attaches to NIC driver to early process incoming packets. `TC` and `XDP` are also part of eBPF infrastructure which eliminates most of security concerns and provides the ability to insert safe bytecode within the OS kernel from a userland program. Experimentations under Linux 5.3.5 *(i)* show VTL ability to quick off the deployment of protocol and to ease their adoption by applications, and *(ii)* evaluate VTL performances by taking legacy TCP stack as reference.**

***Index Terms*—Communication Protocols, TCP/IP, eBPF/XDP, Linux OS, Runtime Deployment, Prototype implementation and testbed experimentation.**

## I. Introduction

THE fundamental role of Internet Transport layer is to ensure end-to-end data moving between applications running on end-systems by providing multiplexing/demultiplexing mechanisms based on the association of IP addresses and port numbers. Additionally and in most cases, it ensures the regulation of data transmission at end-systems as well as within the network. With the growth of the Internet, several Transport layer protocol mechanisms [e.g., 4-11] have been proposed in the literature to improve the end-to-end quality of service with the aim to satisfy the raising requirements of applications and to adapt to a variety of emerging networks.

Unfortunately, as reported in Fig. 1, most of those protocols either are not available within the mainstream OSes and/or suffer from lack of adoption within the Internet. Additionally to data reported at the top of Fig. 1, results of recent studies [1], we capture a dataset of incoming and outgoing Internet traffic to and from two Linux servers running different OS and deployed at our private datacenter. The examination of the data under Wireshark analyzer tool leads us to the same conclusion as [1]: *despite their convenient conceptual approach and better performances against TCP/UDP in several cases [16, 18], any new protocol solutions other than TCP and UDP are (i) faintly deployed (in worst cases un-deployed) and (ii) suffer from shy-of-adoption by applications on the Internet*. This situation prompts sclerosis of the Internet Transport layer also known as ossification of Transport layer that explicates the lack of effective innovation in this layer of Internet communication stack. We argue that this sclerosis of Internet Transport layer is a major barrier to the introduction of more agility and flexibility necessary to the evolution of new paradigms in the Internet such as the emerging softwarized networks and their instantiations [42]. This paper therefore proposes to address this sclerosis.

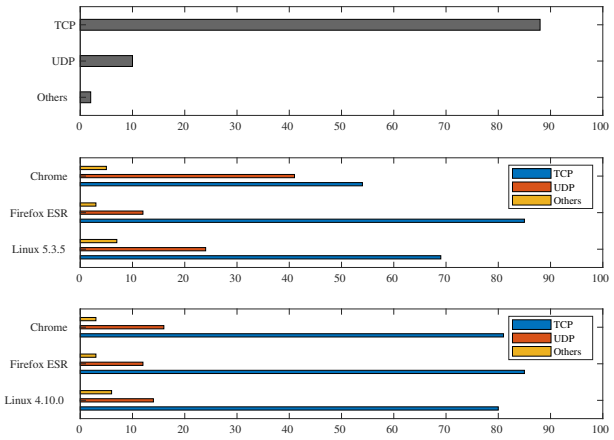


Fig. 1: Transport protocols usage repartition. (1) To navigate on youtube.com or gooogle.com, the same application (Chrome) uses different transport protocols depending on whether it is running under Linux 4.10.0 (bottom) or under Linux 5.3.5 (middle) ⇒ *the used protocol depends on it availability on the OS*. (2) The same kind of application (i.e. web browser, here Chrome and Firefox ESR) surfing on the same website (youtube.com or facebook.com) don't use the same Transport protocol ⇒ *the used protocol depends on the one adopted by the application programmers*.

Deal with challenges (see section II) associated with the Transport layer sclerosis calls for resolving two major questions: *(i) How to deploy in a timely manner a specific protocol mechanism at the end-system? (ii) And, assuming the availability of the mechanism on the end-system, how to ensure its*

E.-F. Bonfoh, S. Medjiah, D. C. Tape, and C. Chassot are with the LAAS-CNRS laboratory, University of Toulouse, 31400, Toulouse, FRANCE (e-mail: name@laas.fr).

*seamless usage by (existing) applications?* The response to this last matter is a key factor in the stimulation of the adoption of any new protocol mechanism as long as this protocol meets the needs of the application. However, most of the prior research efforts to bring extensibility at the Internet Transport layer focus on only one of the two above-mentioned concerns, not yet both till now as far as we know.

On the one hand, part of investigations [2, 15, 28, 29] to address Transport layer sclerosis proposes to rethink the *whole layer architecture* in a way to eliminate current socket API limitations. This approach goal is to ease the adoption of all protocols available on the end-system by *non-legacy* applications. Nevertheless, this approach presents two main drawbacks. First, it makes the assumption that the required protocol is already available on the end-system OS and therefore lacks a method to dynamically deploy a new protocol mechanism when it is not available at the end-point. Second, it doesn't provide any transparent support for *legacy* applications, these should be rewritten to leverage the new architecture.

On the other hand, several studies build their approach around a *specific protocol* by either *(i)* extending an existing protocol (e.g. TCP) [6, 19, 22, 23], *(ii)* building a part of the protocol in user-space on top of UDP [7, 22, 27], or *(iii)* by moving the entire network protocols in user-space [17, 24, 25, 26]. The studies that follow such an approach have the worth of showing how to accelerate the deployment and the extensibility of a kind of specific protocol. Nevertheless, such an approach lacks a protocol-independent API and then forces applications to bind to a specific protocol at their design-time and may, therefore, leads to slow adoption.

In the light of the above observations, this paper argues there is still a need for an approach that jointly *(i)* treats the *deployment* issues within the end-systems and *(ii)* enables the acceleration of the *adoption* of new protocol solutions by applications. Our approach follows the below design principles:

***Transparent integration of legacy applications***. As we will see in section II, most of the existing applications use the standard API socket to consume Transport layer services. Most of the time, application programmers are unwilling to switch from the standard API socket to the API of any new architecture. Therefore, to ease and stimulate its adoption, the Transport layer must provide transparent support to legacy applications i.e. those applications should consume Transport layer services without the need to rewrite them.

***Separation of protocol from aware-application***. In the current standard socket API, the legacy application specifies the protocol to be used at the design-time. This leads to a binding and dependency of the application to a unique and specific protocol preventing the timely structural adaptation of the protocol to the evolution of network state or to the change in the application requirements. Our approach breaks this static tie between the application and protocol by providing a *protocol-agnostic* API to *aware*-application in the way that the latter expresses its needs (in terms of Transport features associated with QoS parameters) instead of the specification of the protocol to use. This principle is in the same line with the TAPS standard [2] but goes far by providing seamless support to legacy applications as stated above in the first principle.

***Protocol modularization***. In order to ease per-session protocol configuration and reconfiguration, we split any protocol in a set of small units called Transport Function (TF) that are packaged in deployable software wrappers (such eBPF programs, loadable kernel modules or Docker containers). Further, the fact to integrate this principle brings to our approach most benefits of modularization namely *(i) the reusability*, i.e., folks other than the TFs developers are able to use them in order to compose another protocol without the need either to know or to change the code inside those TFs; *(ii) the adaptability*, i.e., the ability to adapt to the evolution of network state or applications requirement by replacing or inclusion of new TF that provide more appropriate features in the new conditions; and *(iii) the customization/efficiency*, i.e., the capacity to tailor the protocol to the application requirements and therefore reduce overheads by the elimination of unnecessary services.

We design and implement our approach in VTL, a protocol deployment and data delivery management system that, from a userland library performs *runtime deployment* of protocol mechanisms within the OS kernel.

To realize its goals, VTL leverages and combines two kernel subsystems: `XDP` and `TC`, part of eBPF framework. The aim of this association is *(i)* to grasp the outcoming packets as late as possible just before they reach the Tx NIC ring buffers and *(ii)* pick up the incoming packets as early as possible before they reach the legacy network stack. Therefore, on its egress path, VTL places `TC` hook under the IP layer in order to process all outcoming packets as soon as they left the kernel network stack. To lower as possible the legacy network stack overheads on transmission, VTL takes advantage of RAW sockets to send applications' payload data directly to the IP layer without any L4 processing. On its ingress path, VTL attaches `XDP` hook to the Rx NIC driver to get and process the raw incoming frames as soon as they enter the NIC without letting them reach the legacy network stack. The goal is to fast deliver the data to the application and completely remove legacy network stack processing overheads at the reception. The fact that `TC` and `XDP` are incorporated into eBPF infrastructure allows using its verifier to guarantee the safety of the deployed bytecode and then reduce most of security concerns and end-system crash risks.

The main contributions of this paper are summarized as follows: *(i)* we propose a technique to 1/ timely deploy protocol mechanisms within the OS kernel and to 2/ ensure their transparent use by aware-application as well as legacy one; *(ii)* we design and implement VTL, a protocol deployment and data moving management system; *(iii)* we implement from scratch a set of protocol mechanisms to demonstrate and test VTL capabilities to deploy and reconfigure at runtime protocol mechanisms; *(iv)* finally, we conduct thorough experimentations of VTL under Linux 5.3.5 to evaluate its performances (in terms of latency and throughput); we take Linux legacy TCP as a reference of performance evaluations.

## II. VTL Design Space

This section will discuss the Transport layer sclerosis causes and the resulting requirements for the VTL system. The goal

is to present the ecosystem of VTL and to introduce an overview of its keys components. An insightful inspection of the literature [3, 15, 17, 21] leads us to identify three causes of the sclerosis of the Transport layer. The first two are related to end-systems and are the focus of this paper. The last one falls outside the scope of the contributions of this paper (see section I).

### A. Deployment Barriers

OS constitutes the location environment of the Transport protocol mechanisms. The literature review shows that at the question – *in what space of the OS the protocol must be implemented and executed?* – two main approaches subsist. On one side, for security and performance considerations, several authors suggested to implement and execute the protocol mechanisms in the kernel-space of the OS [32, 34, 35, 40], and most of them went further and proposed to offload these mechanisms in the NIC driver of the end-system [12, 36, 37, 41]. On the other side, a series of studies proposed to move protocol stacks in the user-space [17, 24-26] of the OS with the goal to gain more flexibility and to ease the integration of new protocol solutions. More recent works in line with this latter approach leveraged software acceleration tools such as DPDK [13] or NetMap [14] to reduce performance degradation induced by the execution of protocols within a user-space program. Nevertheless, user-space protocol deployment leaves security and isolation concerns on the table [31].

Altogether, the appearance of software acceleration tools such as DPDK and Netmap is *relatively* recent and the choice that has since been made by OS developers is to dedicate the kernel to the implementation and execution of Transport protocol components. This choice is not without consequence. To provide support for a new protocol, OS developers must integrate it into the kernel, which requires upgrading of their system. OS upgrade is not only time-consuming, very tedious and error-prone, but it also poses enormous software and hardware compatibility issues. As a result, OS upgrade frequency is slow and for OS developers, only a high demand from application developers can motivate and quick-off the integration of a new protocol solution.

As a result, VTL must provide a ***safe and isolated runtime environment*** for protocol mechanisms in the way that the integration of new protocol components is transparent to the OS and has little or no impact on it. This should not be obtained at the sacrifice of flexibility. To fulfill those requirements, we opted for the in-kernel *eBPF VM* as the runtime environment for protocol functions. Thanks to its integrated verifier, eBPF VM provides necessary isolation and safety: each protocol component is checked before its insertion in the VM to guarantee that its execution will not harm the OS. Furthermore, eBPF infrastructure provides to VTL the ability to introduce programmability within the OS kernel by enabling on-the-fly user bytecode insertion and temporary modification of kernel behavior: protocol components may be inserted in a timely manner without the need to recompile and upgrade the OS.

### B. Shy-of-adoption Explanation

The main interface used by applications to consume the services provided by the Transport layer is the standard *socket API*. This API has the particularity of requiring application programmers to choose the protocol at the design time of the application (i.e. when the code is written). Therefore, to adopt any new protocol solution, programmers must modify their applications. This can quickly become complex and is a source of instability if it is necessary to rewrite application each time a new protocol solution is released and best matched the application needs. As a consequence, application programmers prefer to rely on standard protocol solutions 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, even if this one is more appropriate to meet the QoS requirements of their applications.

It results that VTL must provide a ***protocol-agnostic interface*** to the applications. In this work, the protocol-agnostic interface is similar to TAPS API [2]. In this way, applications don't have anymore to invoke a specific protocol but just specify the services they want from VTL. The choice of the most appropriate protocol mechanisms to satisfy application needs is left to VTL. Furthermore, VTL must support legacy applications without the need to rewrite them. This may be performed by seamlessly redirect legacy applications socket API invocations towards the protocol-agnostic interface.

### C. Other Barriers Out of End-systems

Indeed, while the logic that prevailed at the birth of the Internet is based on an end-to-end principle where the network, composed mainly of routers, had no visibility over what happens beyond the L3 level, the massive introduction of middleboxes such as NAT, IPS/IDS, firewalls, proxies, etc., has "violated" this principle. Those middleboxes are able to read and modify packets up to level L4 and above, and reject any unrecognized protocol. For security and performance reasons, middleboxes vendors are unwilling to configure their devices to whitelist any new protocol until it is supported by the most popular OS developers. Nevertheless, through extensive measurements, the authors of [17] 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 place focus on the end-systems and we preconize systematic fallback to TCP or UDP to prevent complete failure in case of rejection during middlebox traversal.

## III. VTL System Overview

Geared by our analysis (see section II), we designed and implemented VTL, a system that is able to timely *deploy a protocol component* within the OS kernel and ease their *adoption* by providing protocol-independent API to aware-application and seamless support to legacy ones. As stated in section I, VTL is designed around a set of principles namely the notion of a *protocol composition from a set of basic functions*. This section introduces the key concepts of this principle and presents an overview of services and features

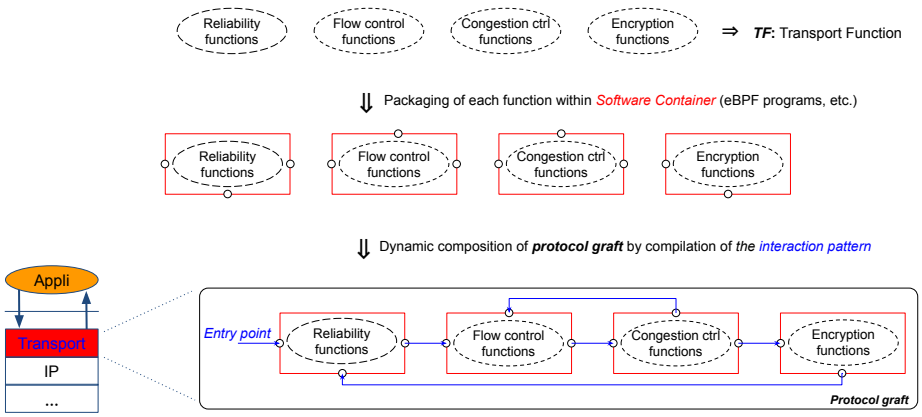


Fig. 2: Tranport Function (TF) and Protocol Graft illustrations.

provided by VTL. It is worth to note that the concepts related to a protocol composition are not new and specific to VTL; therefore, they are not present here as a contribution of this paper but only to show up the readers the way VTL organizes its data plane for more efficiency in management of protocol deployment and data delivery. The main contributions of this paper are those stated in section I.

### A. Transport Function (TF) and Protocol Graft 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 provides one *property* that could roughly be grouped as follows (see Fig. 2): reliability functions, flow control functions, congestion control functions, security/encryption functions. Readers should note that all functions related to connection management (opening, parameters negotiation, teardown, etc.) are provided by default within the control plane of VTL by *Control Broker* component (see Fig. lambda). Each TF must be made "loadable" by it wrapping in software container (eBPF programs, in the current implementation of VTL) that will provides it with at least two important interfaces: (1) the *input interfaces* from which it should receive any packet data or control information; (2) the *output interfaces* to which it should push any packet data. Furthermore, TF should have access to a *shared memory* created and maintained by the protocol graft to which it belongs.

*2) Protocol Graft:*

As illustrated in Fig. 2, a *protocol graft* is a list of loadable TFs with their *interaction pattern* defining the way the TFs are connected. It should provide one or more *services* such as reliability service, ordering service, multipath service, encryption service, and so on. A protocol graft is distributed between the sender end-system and receiver ones and provides a comprehensive single or more Transport feature(s). The sender side maintained the *egress graft* whereas the receiver side maintained an *ingress graft*. Fig. 3 illustrates a protocol graft distribution for a typical ARQ-based reliable service; note that this is only for illustration purposes; more comprehensive reliable service should have many other functions such as duplication control. 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.

### B. 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. To do that, VTL priorly provides applications with an access point to Transport services in order to allow it *(1)* to express it requirements, *(2)* to send and receive data, and *(3)* to reset it requirements on-the-fly. The access point is explicitly provided from a *protocol-agnostic* API in the case of aware-application and seamlessly attributed to a legacy application so that the latter should consume VTL services without any modification. From an internal point of view, VTL ensures the Transport session establishment as needed; ensures the selection, the negotiation, and the deployment of the right protocol graft; and monitors the network state in order to adapt at runtime to the change of network 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. Finally, VTL provides protocol developers with

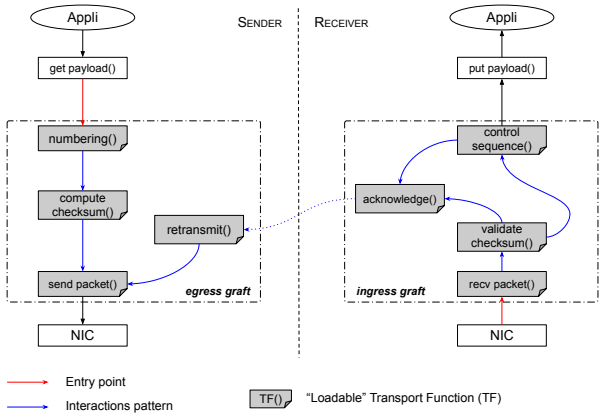


Fig. 3: Illustration of protocol graft providing an ARQ-based reliable service.

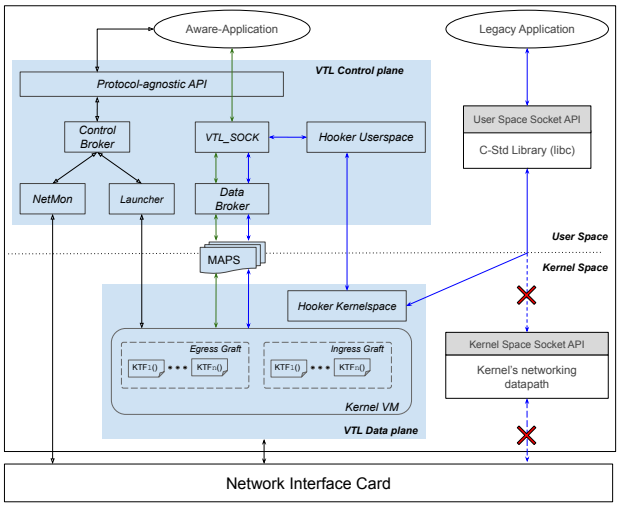


Fig. 4: VTL System Architecture Overview. Blue arrows represent the legacy application datapath, green arrows the aware-application datapath, and black arrows the KTFs deployment path.

the facilities to write, test and publish new protocols within the end-systems. The rest of this subsection provides the specifications of QoS and Transport Features in the context of VTL.

*1) QoS and Transport Features:* VTL allows applications to express their requirements by firstly characterizing the desired Transport services and then associating to those Transport services a set of required or desirable quality of service parameters. A requested Transport feature should be expressed in terms of *reliability* (full or partial), *order* (full or partial), *flow control*, *congestion control*, *security*, and *multipath*. Additionally to these Transport features, VTL allows applications to associate QoS parameters that can be expressed in terms of maximum acceptable *delay* or *latency* (`max_delay`, expressed in ms), minimum *throughput* (`min_throuphput`, expressed in Mb/s) and allowable *loss rate* (`max_loss`, expressed as a percentage). It should be noted that certain services requirements by applications may be redundant, such as an application that requests to set its allowable loss rate (`max_loss`) to zero and at the same time invokes the use of a fully reliable Transport service. In the current implementation of VTL, the selection of the most appropriate TFs to meet the application requirements is based on predefined matching rules built on top of a set of decision trees. Future implementations of VTL would include more complex selection models based on machine learning techniques.

## IV. VTL Design and Implementation

### A. KTF: and eBPF Instantiation of TF

*1) eBPF Overview:*
The goal of this section is to provide the reader with a background on eBPF [43]; the core technology on top of which VTL is currently built. The authors of [44] presented the Berkeley Packet Filter (BPF) virtual machine as a kernel agent aiming to capture as early as possible incoming packets on the host's network interface card (NIC). For a long time, BPF has been used for packet filtering. Recently, Linux introduced an extended version under the name of eBPF [43], for extended BPF. The name of eBPF no longer reflects the reality and is confusing because a plethora of new features are added to it, and its usage scenarios go far beyond the naive filtering. Indeed, eBPF is used today by many industrials like Facebook or Netflix to perform tracing, monitoring, networking or to enhance the security of their systems. As its predecessor, eBPF allows injecting bytecode within the OS kernel at runtime, i.e. without having to recompile the OS. We can summarize eBPF features into three major components: *maps*, *helper functions*, and *tail calls*.

| *Helpers* | *Description* |
|---|---|
| `vtl_start_timer(i, n)` | Set timer i to n *ms* |
| `vtl_stop_timer(i)` | Stop a timer i |
| `vtl_build_graft()` | An exogenous wrapper of `bpf_tail_call()` helper |

TABLE I: Main added helper functions within VTL System.

*Maps* are generic 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 are always attached to the root file system `/sys` and therefore are very useful to ensure data persistence between successive invocations of eBPF programs.

Access to kernel functions by eBPF programs is strictly controlled to prevent OS damage; consequently, the eBPF program has a list of accessible functions called *helper functions* that are part of the kernel. These helper functions allow eBPF programs to interact directly with the kernel. For instance, to perform tail calls, `bpf_tail_call()` helper may be used by the current eBPF program to ask the kernel to jump to another eBPF program. Tail calls feature can be used to build a chain of eBPF programs and therefore to implement modularization.

Each deployed eBPF program inside the OS kernel must be attached to a *hook point*, a.k.a. as a *kernel event* (incoming packet, system calls, socket operations, ...). Then, each time the event occurs, the eBPF program attached to it is executed. eXpress Data Path (`XDP`) and Traffic Controller (`TC`) are the two main networking hook points around which VTL is built.

*2) KTF Specification:*
KTF, for *Kernel Transport Function*, is an instantiation of TF in the form of eBPF program. As such, it inherits common properties of an eBPF program, i.e.:

- ***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 as soon as they arrived at the NIC driver);
- ***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 connection negotiation state.
- ***Output interfaces***: *helper functions* used to implement core algorithms and serving as output points towards either the application or the network.

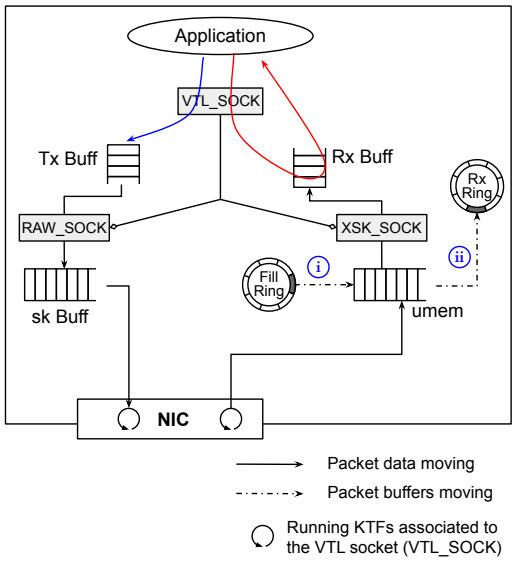


Fig. 5: Data moving between application and VTL Socket and its associated buffers.

It is worth to note that VTL extends helper functions provided by the native eBPF VM in order to address some specificity related to protocols functioning. For instance, a simple stop-and-wait protocol needs a timer to trigger packet retransmission in case of loss. But 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 1. summarizes the set of added helper functions by VTL at the writing time of this paper.

### *B. VTL Design*

The main components and workflows of VTL are shown in Fig. 4. VTL components are separated between two planes: the *control plane* constituted by a userland library and the *datapath plane* constituted mainly by the KTFs runtime environment (eBPF kernel VM). Both planes share information (ACK/NACK, KTF negotiation state, packet data, ...) thanks to a set of shared `MAPS`. As already stated, those `MAPS` are attached to the root filesystem `/sys` and additionally serve as a buffer of deployed protocol grafts. VTL defines two main workflows: *KTF deployment workflow* and *data delivery workflow* that are described more in-depth below.

*1) KTF deployment:*

KTF deployment workflow defines how VTL deploys a new protocol function within the eBPF VM and attaches it to the right hook (`TC` or `XDP`) depending on whether the KTF is intended to process incoming and/or outgoing packets. KTF deployment is decided by *Control Broker* component and can be triggered by three events: *(i)* an invocation of the regular API or the protocol-agnostic API by applications, *(ii)* a request for a connection from remote end-system received on VTL socket (section IV.B.2), and *(iii)* a change in network condition reported by the *NetMonitor* daemon. When *Control Broker* requests the deployment of a specific KTF stored as a user-space object file, *Launcher* component picks it in *KTFs pool* and starts its loading within the eBPF VM. *KTFs pool* is a repository of a set of precompiled and ready to deploy KTF. The precompilation of KTF in object file eliminates Clang/LLVM compilation overheads during the deployment of protocol functions (see section VI). Prior to the effective loading of the KTF, it is checked by the verifier that performs a series of verification such as *(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; and *(iv)* the number of instructions in the KTF that must be under 1M (4096 for all Linux version prior to v5.2.0). The goal of verifier operations is to ensure that the deployed KTF will not crash the OS kernel. As soon as 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 it is attached to the `XDP` hook and all outcoming packets if it is attached to the `TC` hook.

*2) Data delivery:*

Data delivery workflow determines how application payloads and protocol headers are moved on egress/ingress paths by VTL framework in order to ensure optimal data transfer between end-systems. Application payloads transfer may be preceded by protocol graft negotiation stage (section IV.B.4) that ensures that the appropriate KTFs to satisfy application needs are deployed and ready to process incoming and outcoming traffics. *Data Broker* controls data moving by configuring and providing to application a VTL socket that is the access point to VTL Transport services. VTL socket emulates either a RAW socket for data transmission and/or an XSK socket for data receipt (Fig. 5). After obtaining a VTL socket (section IV.B.3), a VTL-aware application is able to use the protocol-agnostic interface to send and receive their data. *Hooker* component provides transparent support to legacy applications; this support may be activated or not. When the system admin activates this support, it systematically enables *Hooker* component to interface to regular socket API. Therefore, legacy application send/receive operations are seamlessly redirected by *Hooker* towards VTL datapath (section IV.B.6).

Fig. 5 depicts how internally the VTL socket transfers data through its associated buffers; for clarity purpose, *Data Broker* and *protocol-agnostic API* are not shown in this Figure. During data moving, the interactions between the application and VTL system are performed asynchronously thanks to a peer of buffers at the transmission (*Tx buff* and *skb buff* in Fig. 5) as well as at the reception (*Rx buff* and *umem* in Fig. 5). *Rx buff* and *Tx buff* are useful to ensure the protocol reconfiguration without interrupting application. 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, there is no intermediate IP layer processing and the received data may be sent directly to *Data Broker* in userland for fast delivery to the application. With the aim of

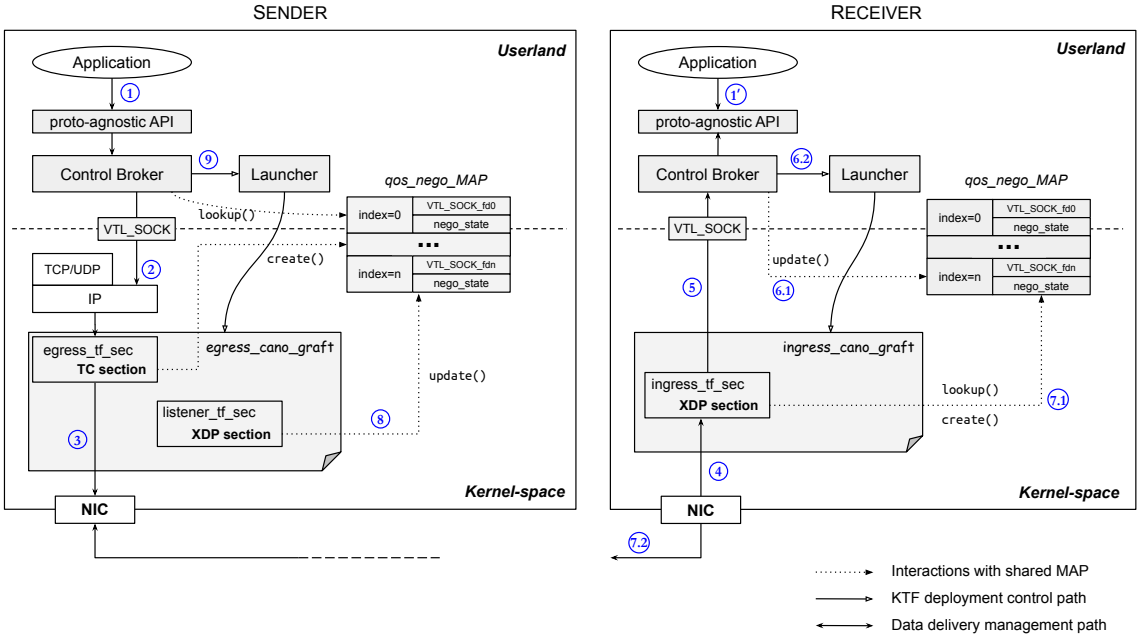


Fig. 6: Protocol grafts negotiation process under VTL System.

making use of XSK socket zero-copy capability, *Data Broker* and the OS kernel share the *umem* buffer. Therefore, to avoid memory access conflicts and deadlock issues, VTL leverages AF_XDP socket family [39] features to associate two ring buffers to the umem: the *fill ring* that is used by *Data Broker* to pass the ownership of packet buffer to the kernel (ⓘ in Fig. 5) and the *Rx ring* that is used by the kernel to pass the ownership of packet buffer to *Data Broker* (ⓘⓘ in Fig. 5). 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 securely 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*.

*3) Obtaining VTL Socket:*
In order to transmit and receive data, the application must first obtain a VTL socket. The application indicates its transfer mode in one of the following three modes: sender, receiver, or both. Based on the transfer mode specified by the application, *Control Broker* creates and configures a new VTL socket and its associated buffers (Fig. 5) and triggers the deployment of the ***canonical graft*** (see section V.A.1). Finally, *Control Broker* associates the file descriptor of the deployed canonical graft 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 can use to send and receive its data. The canonical graft aim is to allow the application that doesn't require a specific QoS need to directly send and receive its data without additional overheads and unnecessary delays. For application with specific QoS requirements, the canonical graft is used to conduct the KTFs and protocol graft negotiation stage. Additionally, the *ingress* canonical graft is useful to conduct a stateful runtime reconfiguration of protocols (section V.A.5).

*4) Protocol Graft Negotiation and Deployment:*
The protocol grafts negotiation process between a sender and receiver is shown in Fig. 6 and in case of successful negotiation, it ends up by the deployment of the suitable grafts to satisfy application requirements. Each side of the connection maintains its own map named `qos_nego_MAP`. Each index of the `qos_nego_MAP` contains the file descriptor of the VTL socket and the associated graft negotiation outcome (`N_ACCEPT` / `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 side: the client of the negotiation***. VTL-aware application that requires specific QoS defines it by invoking the protocol-agnostic API ①. Based on a set of 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 by setting up its *gid* header field especially useful to tell the receiver the specific ingress graft it must deploy. Once it finishes packet preforming, *Control Broker* transmits it to the IP layer ② and 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. As soon as the packet left the IP layer, it is intercepted by `egress_tf_sec` which based on *gid* field may determine if the intercepted packet is a negotiation one or not. When the *gid* value is not set (value is, in that case, `NULL`), the packet contains application payload and

it is not a negotiation packet. Therefore, `egress_tf_sec` sets packet *type* value to `DATA`. Otherwise, it is a negotiation packet and then `egress_tf_sec` sets the packet *type* value to `NEGO`. The packet is finally transmitted to the network device ③. The negotiation acknowledgment packet, sent by the receiver 7.2 in reply to negotiation request, is intercepted and processed by `listener_tf_sec` which updates the `qos_nego_MAP` to signal to *Control Broker* the receiver response to negotiation request ⑧. In the case of acceptance (value set to `N_ACCEPT` on the map), *Control Broker* triggers the deployment of the previously selected egress graft to replace the canonical one ⑨.

***Receiver side: the server of the negotiation***. After the application has finished obtaining a VTL socket (see IV.B.3), it issues a blocking request to the protocol-agnostic API 1' to expect and to retrieve the outcome of the negotiation handled by *Control Broker*. At the reception of a *negotiation* packet from the sender ④, `ingress_tf_sec` delivers it to *Control Broker* which extracts the *gid* value of the packet 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 deployment of the requested ingress graft that provokes systematic deactivation of the ingress canonical graft 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 take 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 it does not.

*5) Protocol-agnostic API:*

*Protocol-agnostic API* is a shared library used by VTL-aware applications to express their QoS requirements and to send and receive their data. The *protocol-agnostic API* is easily extensible and may integrate other functions in the future to respond to more extensive use cases. We describe the currently implemented functions through a typical function call flow as illustrated in Fig. 7. Furthermore, Table 2. summarizes the way application should specifies parameters when using this API.

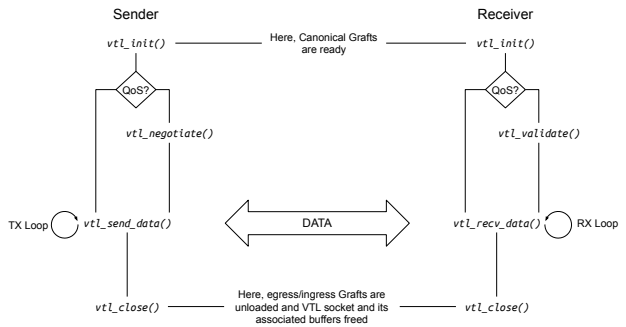


Fig. 7: Typical Function call flow by VTL-aware applications for data Tx/Rx.

| Function | Params | Description |
|---|---|---|
| `vtl_init()` | mode, src ip, dst ip | Create a new VTL socket (buffers, canonical grafts, etc.) |
| `vtl_negotiate()` | vtl_sock, qos_values | Get sender application QoS requirements and trigger a graft neotiation process |
| `vtl_validate()` | vtl_sock | Retrieve graft negotiation outcome to receiver application |
| `vtl_send_data()` | vtl_sock, buffer, buffer_size | Send application payload data and retrieve the size of written data |
| `vtl_recv_data()` | vtl_sock, buffer, buffer_size | Fetch application payload data and return the readed data size |
| `vtl_close()` | vtl_sock | Close VTL socket and free it associated resources (buffers, KTFs, etc.) |

TABLE II: VTL protocol-agnostic API functions parameters.

Prior to any request, the aware application must call into `vtl_init()` function that will trigger the creation and configuration of a new VTL socket (section IV.B.3). At this stage, thanks to canonical grafts *associated* with the newly created VTL socket, the application (without any special QoS) may directly enter in its Tx/Rx loops to start data transfer by issuing `vtl_send_data()` or `vtl_recv_data()`. In contrast, before entering in its Tx/Rx loops, a sender application with specific QoS requirements may call into `vtl_negotiate()` to transmit their needs to VTL system. At the receiver side, application must invoke the blocking function `vtl_validate()` to indicate to VTL system that he is waiting for a graft negotiation stage result. Both functions `vtl_negotiate()` / `vtl_validate()` should return a positive value (the file descriptor of the newly deployed graft) to signal a successful negotiation to the application or a negative value in case of failure. At the end of data transfer, applications should issue `vtl_close()` function to close the file descriptor associated with VTL socket; this will *(i) free* the buffers associated with the socket as well as *(ii) unload* the KTFs of grafts associated with this socket.

*6) Hooker: Legacy Application Support:*

*Hooker* component's goal is to provide support to legacy applications. In its current implementation, VTL supports all regular applications based on TCP. As shown by results reported in Fig. 1, since almost 90% of Internet applications are based on TCP, we argue that support to TCP applications may be a significant step to accelerate the adoption of regular applications. The internal structure of Hooker is shown in Fig. 6.

When sysadmin activates support for legacy applications, *Hooker* attaches to the root `cgroupv2` [20]; therefore, by making use of the hierarchical model of cgroups, it processes every ingress and egress packets of all processes running on

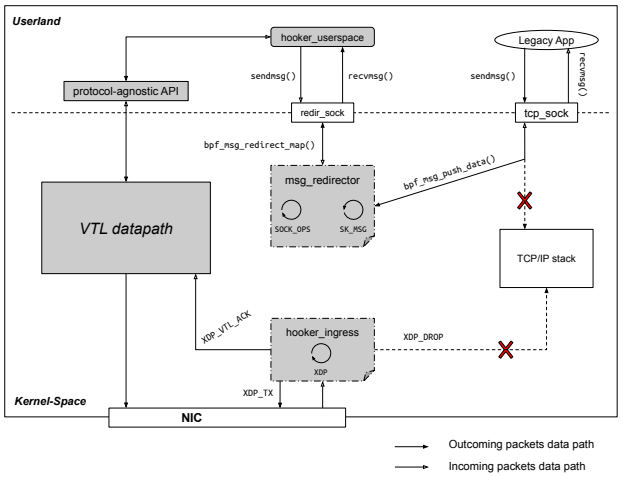


Fig. 8: Legacy applications data moving under VTL System.

the end-system. *Hooker* maintains a map of type `SOCKMAP` that key is a struct 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 packet data by calling into `sendmsg()` on a TCP socket, the `SK_MSG` bpf program section running by *msg_redirector* intercepts it. When *Hooker userspace* invokes `recvmsg()`, it receives only the payload without Transport layer address information. By consequence, in order to allow the *Hooker userspace* to deliver data to the right process when it receives a response from remote peer end-system, before making data available on the redirection socket, *msg_redirector* rewrites it by adding one tag field in the header thanks to buffer extension permitted by the helper function `bpf_msg_push_data()`. Finally, to deliver the message either to the redirection socket or 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, as soon as the host NIC driver receives packet data, the `XDP` bpf program section running by *Hooker ingress* intercepts the packet data 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 and acknowledgment of the packet.

## V. Use Cases

### A. Carried Out Use Cases

For proof-of-concept and testing purposes, we revisit and implement, *from scratch*, a set of protocol mechanisms well-known in the literature. All implemented mechanisms have the same structure. At the sender side, the egress graft maintains two KTFs: a `TC` section named `egress_tf_sec` that ensures the processing of all outcoming VTL packet, and an `XDP` section named `listener_tf_sec` that job is to process acknowledgment packets. At the receiver side, the ingress graft runs a single KTF composed by an `XDP` section (named `ingress_tf_sec`) sufficient to process incoming packets and to send acknowledgment if needed thanks to the verdict `XDP_VTL_ACK`. This section presents those mechanisms.

*1) Egress/Ingress Canonical Grafts:* Canonical grafts, egress one as well as ingress one, purposes are *(i)* to enable the immediate transfer of data of applications that don't need a specific QoS and *(ii)* to conduct the KTFs negotiation stage for QoS-oriented applications. Additionally, the *ingress* canonical graft is useful to ensure the reconfiguration of protocols (section V.A.5). Both canonical grafts are deployed at the creation of a new VTL socket to which the KTFs composing those canonical grafts are associated by default.

At 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 if necessary the updating of `qos_nego_MAP` (Fig. 6) to signal to *Control Broker* the receiver reply. 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 the processing of the next incoming packet; otherwise (the received packet is negotiation one), it waits for the outcome of negotiation handled by *Control Broker*, transmits an acknowledgment to the sender and pursues the processing of the next packet. Moreover, the *ingress* canonical graft has the property to be systematically activated or deactivated when there is no any other `XDP` section running on the NIC or when a new `XDP` section is attached to the NIC respectively. This latter property is essential to ensure a stateful reconfiguration of KTFs (section V.A.5).

*2) ARQ Reliable Graft Based on Stop-and-wait:* Stop-and-wait algorithm is the most basic ARQ-based flow control that ensures loss and error recovery and in-order delivery of data. At the sender part, it consists of sending one packet at a time. At each time there is only one packet in-flight. That packet is, after a while, automatically retransmit if the receiver doesn't positively acknowledge it; otherwise, the next data packet is transmitted.

At the sender side, `egress_tf_sec` KTF creates and maintains a hash map named `egress_socks_MAP` that is regularly updated by `listener_tf_sec` KTF at the receipt of an acknowledgment packet from the receiver. Therefore, after timer expiration, by reading the map, `egress_tf_sec` is able to determine if there is a need for packet retransmission or not. The value of the timer duration must be set depend on the RTT. If its value is under the RTT, `egress_tf_sec` will read the map before the reception of acknowledgment and therefore possibly trigger unnecessary packet retransmission(s). At the receiver side, `ingress_tf_sec` KTF ensures that the received packet is not corrupted and sets the acknowledgment packet *type* value to `ACK`; otherwise, it sets the *type* value to `NACK`

and sends a reply to the sender.

*3) ARQ Reliable Graft Based on Go-back-N:* Go-back-N flow control is an optimized version of stop-and-wait algorithm. Instead of sending only one packet at a time, Go-back-N mechanism allows the sender to transmit at a time $N > 1$ packets without waiting for acknowledgment from the receiver. The aim is to reduce at maximum the idle time of the simple stop-and-wait flow control. Furthermore, to ensure in-order packet delivery, sender and receiver make use of sequence numbers as opposed to stop-and-wait algorithm where there is no need for numbering packets[1]. Therefore, at the receiver, additional to data integrity validation, `ingress_tf_sec` makes sure that the packet is in sequence before positively acknowledge it. Code listing 1 illustrates an example of ARQ Go-back-N. To optimize space, we only show the *template* of the egress graft part of the protocol.

```
#include <stdbool.h>
#include <linux/bph.h>
// Other useful headers and ...
#include <../vtl.h>

// Declare a MAP to store packet for retx
struct bpf_elf_map SEC("maps")
EGRESS_PKT_WND_MAP = {
 .type  = BPF_MAP_TYPE_HASH,
 .size_key = sizeof(unsigned int),
 .size_value = sizeof(struct 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 ***/
}
```

Listing 1: Kernel Transport Function (KTF) Example

*4) Partial Reliable Graft:* Partial reliability concept consists of allowing KTFs not to issue at reception all the packets data 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 (such as multimedia applications). This considerably reduces the transmission delay with less impact on the proper execution of the application.

Fig. 9 illustrated this concept under video streaming where each packet data vehiculates one image. The assumption is made that the loss of one packet data (image) every 10 images is acceptable because not perceptible by the human eye. In first case, only one image (I3) is lost, the packet data containing

[1]Note that the assumption is made here that there is no loss on ACK packets.

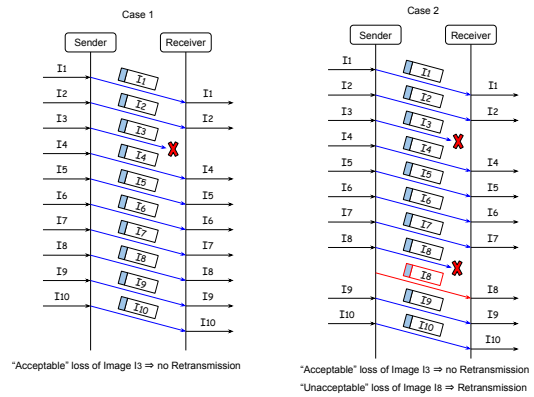


Fig. 9: Partial Reliability illustration under video streaming with accepted loss rate set to *MAX_LOSS* = 10%.

that image is not retransmitted. In the second example, the images I3 and I8 are lost. The first lost packet data is not retransmitted but to respect the *MAX_LOSS* (10% here), the second lost packet data (I8) must be retransmitted.

*5) Runtime Graft Reconfiguration:* VTL system leverages eBPF infrastructure dynamic reloading feature to guide runtime reconfiguration of protocol grafts. It consists of dynamic replacement of KTFs attached to `TC` or `XDP` hooks without application outage. Reconfiguration may be performed either to end up successful KTFs negotiation process (section IV.B.4) or when *NetMonitor* daemon reveals change within the network that requires adaptation actions. At the difference of KTFs negotiation, the reconfiguration trigger by a change within a network doesn't require synchronization between the sender and the receiver, each side conducts its reconfigurations independently. VTL can perform two types of runtime reconfiguration: a *stateless reconfiguration*, to fasten the adaptation actions at the expense of packet loss, or a *stateful reconfiguration*, more conservative reconfiguration approach that ensures that no packet is lost or dropped at the cost of additional overheads and delay, especially at the sender (Fig. 10). The right tradeoff must therefore be found depending on the application use scenario.

During a stateful reconfiguration, on the egress path, VTL leverages the fact that `TC` subsystem allows running more than one `TC` hook at a time on the NIC to load the new egress graft before unloading the old one. In this way, during a while, all packets data transmitted by the application are processed sequentially by both egress grafts. Contrary to `TC` hook, on the ingress path, `XDP` subsystem doesn't allow executing more than one `XDP` hook at a time on the same NIC. To solve this issue, owing to make heavy use of a new map to store *umem* buffers (Fig. 5) before old ingress graft unloading, VTL leverages the systematic activation properties of the *ingress* canonical graft (when there is no any `XDP` section running on the NIC) to ensure the persistence of incoming packets processing. At the loading of the new graft, the canonical graft is systematically deactivated.

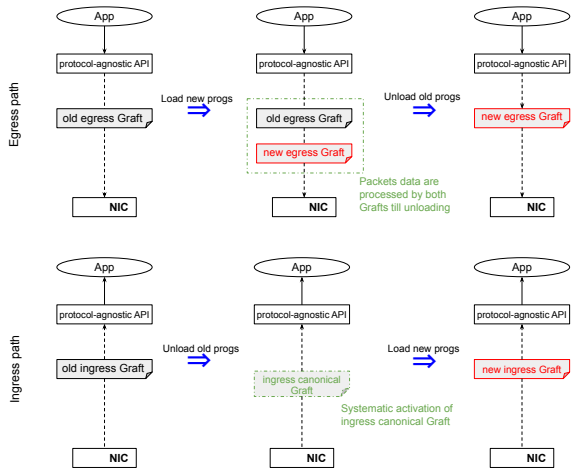


Fig. 10: Protocol Graft stateful reconfiguration actions. To ensure no packet loss, the egress path may generated moderate overheads during stateful reconfiguration.

## VI. Validation and Performance Evaluation

We implemented and evaluated VTL under Ubuntu distribution running Linux 5.3.5. The goals of carried experimentations were to evaluate *(i)* the correctness of VTL, i.e. its protocol deployment and reconfiguration capabilities, *(ii)* the performances of VTL in terms of deployment latency and data sending throughput, *(iii)* the performances under VTL of the implemented KTFs and grafts, and *(iv)* the impact of legacy applications support.

### A. Testbed Setup and Methodology

VTL experimentations 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 [33]. Unless otherwise stated, the network parameters used during experimentations are reported in Table III. Each host is equipped with Intel Core i7-7500U CPUs, 3.8GiB RAM, and Qualcomm Atheros QCA6174 NIC driver.

To evaluate implemented KTFs and grafts, we build two VTL-aware applications, one acting as *data* streaming server and the other one playing client role. In the same way, to evaluate legacy application support by VTL, we build two TCP-based applications (one server and one client too). Each server application is able to stream several kinds of files with different sizes range from a simple 4K file text to more than 16M video files. In order to capture outliers and thus avoid biases, for each metric evaluated, we repeated the experiment enough and observed the standard deviation of the sample. The mean of the observed sample is taken only for a relatively small standard deviation. The reported evaluation of legacy TCP is provided only as a reference for the reader. Therefore,

| min RTT | Bandwidth | Loss rate |
|---|---|---|
| 10ms | 16Mbps | 0-25% |

TABLE III: Testbed Network Configurations.

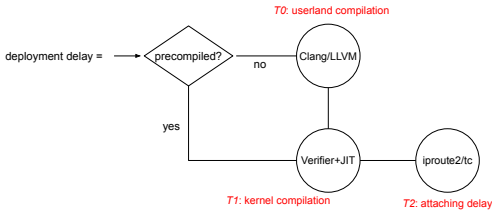


Fig. 11: KTF deployment delay breakdown. Precompiled KTFs eliminate Clang/LLVM compilation delay (T0).

this is not an absolute comparison with TCP since a fair comparison would require a large scale configuration and deployment. The window size of the Go-Back-N and partial reliability grafts is set to 16. Finally, to avoid interference with packets not directed to VTL within the testbed environment, we set IP protocol number to experimental value `0xfd` for VTL packets.

### B. Microbenchmarks

*1) KTF and Graft Negotiation Delay:*
We start by assessing the delay required to complete KTF negotiation process. This delay is composed of *1/* packets processing delay (negotiation one and its associated acknowledgment) and *2/* KTFs deployment delay at both sides (sender and receiver). The experiments carried out demonstrated that the packet processing delay is negligible compared to the KTFs deployment one. Consequently, the results reported in Fig. 12 illustrate essentially the cost of KTFs loading.

As illustrated in Fig. 11, the KTF deployment delay is the sum of the Clang/LLVM userland compilation delay, the verifier operations and JIT kernel compilation and the delay of attaching to `XDP` or `TC` hooks by iproute2 or tc tool respectively. As shown in Fig. 12, we found that *precompiled KTF provides not negligible value*, almost 76% reduction of total delay. Moreover, we note that *the negotiation delay increases with the size of the KTFs to load*, which makes sense in view of the dominance of the loading time of KTFs over the total cost of the negotiation procedure.

The impact of graft negotiation stage should be more or less cushioned by the actual duration of the Tx/Rx of the application payloads. Therefore, a Tx/Rx of short duration will have more interest in keeping the Canonical grafts while for a Tx/Rx of large amounts of data, the application, as necessary, can afford to trigger and wait for the completion of a negotiation procedure.

*2) Moving Data Performances – latency & throughput:*
Each implemented graft is evaluated under the streaming of files of different sizes and under various network conditions, especially loss rate variation. The results, reported in Fig. 14, shows the file completion time (FCT) latency and the speed rate of sending data. We define FCT as the amount of time elapsed between the first packet *sent* and the last packet *received*. The accepted loss rate in the partial reliability graft is set to 10% during experiments. In an ideal lossless network, only the sending of large files allows capturing

the sub-performance of stop-and-wait mechanism. This sub-performance is completely hidden during the Tx/Rx of short files where stop-and-wait graft can align with more elaborated strategies such as Go-Back-N and partial reliability. However, contrary to what one might think, when network conditions deteriorate, a simple stop-and-wait graft has about 1.5x times better throughput than Go-Back-N and partial reliability grafts. This is explained by the fact that the stop-and-wait almost never floods the network with packets, which plays to its advantage in case of high loss.

*3) Legacy Application and Reconfiguration Use Case:* In this section, we demonstrate that VTL can perform a runtime reconfiguration of protocols following a predefined reconfiguration rule and that legacy applications are able to leverage VTL features without modification. The metric under evaluation by *NetMonitor* daemon during this test scenario is the RTT that is reported with 0.5 ms periodicity. When $RTT < 300ms$, VTL deploys and maintains the use of Go-Back-N graft and as soon as $RTT >= 300ms$ it systematically triggers the replacement of Go-Back-N by partial reliability graft with *MAX_LOSS* rate set to 20%. In order *not* to distort the evaluation of support for traditional applications, the loss rates considered in the following scenarios are zero.

The file used in this use case is a 16M video with a duration of 45 seconds. For each of the applications evaluated, legacy and aware, the scenario is identical. During the first 14 seconds, we send the packets under an RTT of 20 ms. From the 15th second, we introduce an RTT of around 320ms and observe the adaptation actions implemented by the VTL. The results reported in Fig. 14 show that the cost of support for traditional applications, observed over the duration of the video is more marked under a low delay network. Over long delay network conditions, this cost is compensated by the gains of partial reliability graft deployed by VTL. As a reference, we

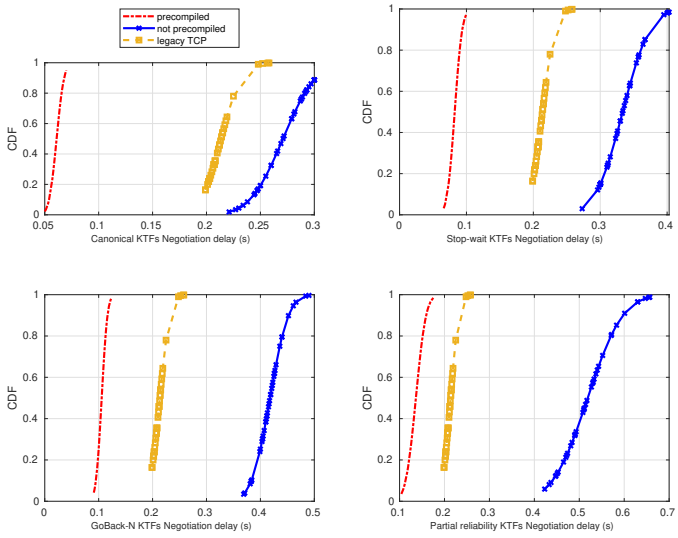


Fig. 12: KTFs and graft negotiation delay. Precompilded KTFs (red) considerably reduce negotiation delay.

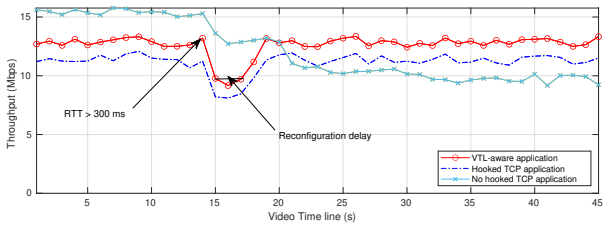


Fig. 13: Runtime reconfiguration benefits and overheads for aware and legacy applications.

observed that the same TCP application without VTL support will show significant degradation of its throughput under long delay conditions.

## VII. Discussion and Related Work

In section I, we have introduced prior work aimed at improving the Internet network protocols stack. These works are numerous; therefore, in this section, we bring up the discussion on the most recent ones.

PQUIC [22] introduced a prototype of a framework able to dynamically extend QUIC [7] by loading at runtime new protocol plugins that contain the mechanisms implementation. In [19], the authors present an eBPF-based framework able to allow user applications to quick off the adding of new TCP options. As with PQUIC, [19] keeps the binding to a specific protocol that prevents the adoption of new functionalities from applications not write for the specific protocol or its new versions. Multistack [17] developed a kernel module able to deploy and manage network protocols in user-space. The main goal of Multistack is to accelerate the deployment of protocols. However, we found during our work that kernel modules can easily crash the whole end-system OS and therefore pose a severe security concern. Furthermore, Multistack requires the rewrite of applications. To overcome this latter limitation, StackMap [25] introduced support for legacy applications. Nevertheless, this support is not transparent for applications.

At the best of our knowledge, NEAT [38] is the single prior implementation of TAPS-like API that permits applications to express their needs without specifying a protocol to use at their design-time. While it eases the way applications consume Transport layer services, NEAT doesn't provide any technique to deploy at runtime a new protocol when it is not available on the OS; the choice is dependent on the end-system OS network stack.

## VIII. Conclusions

We present and implement VTL, a system that is able to timely deploy protocol functions within the OS kernel and ensures their transparent usage by applications. VTL provides seamless support to legacy applications, i.e., their modification is not required to fully consume VTL services. We build VTL on a combination of `XDP` and `TC` Linux subsystems, part of eBPF infrastructure. This combination ensures that packets data are grasped as soon as they left the IP layer and picked up once they arrived on the NIC driver.

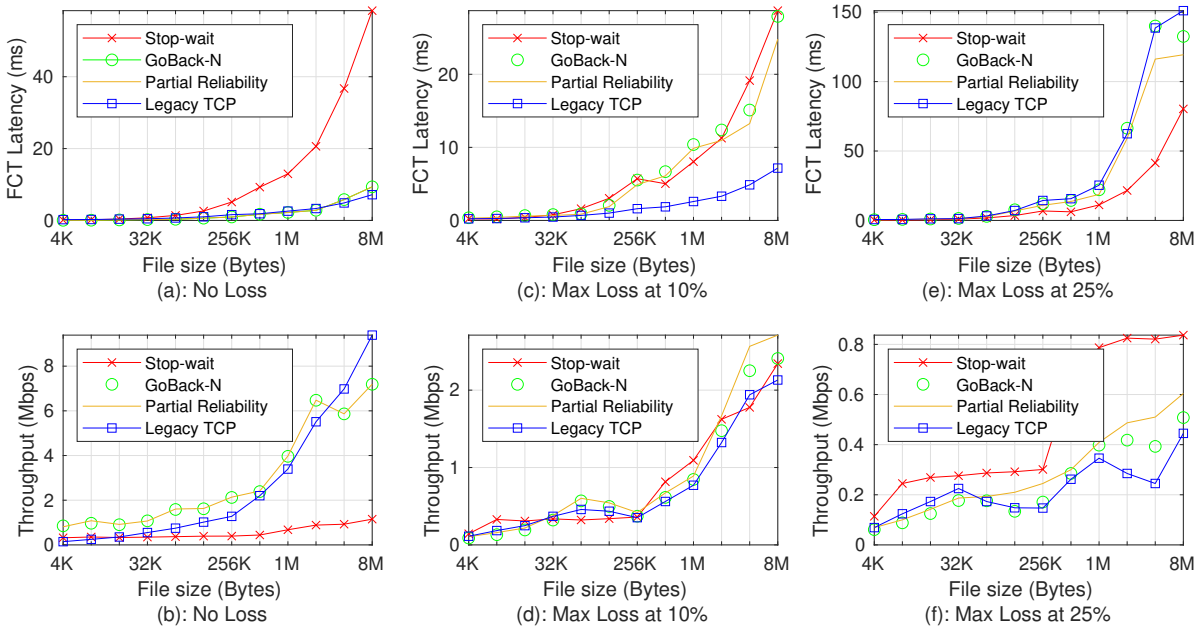


Fig. 14: Data Moving latency (a, c, e) and throughput (b, d, f) under various network conditions.

To evaluate VTL, we implemented from scratch a set of protocol mechanisms. We found during evaluations that a support to legacy applications generated a moderated overhead that must be mitigated by the benefits of VTL. Further, the evaluation of the current implementation demonstrates the runtime protocol deployment and reconfiguration property of VTL and shows that it presents low deployment delay, especially when the deployed protocol functions (KTF) are precompiled and stored in a dedicated repository.

## REFERENCES

[1] D. Murray, T. Koziniec, S. Zander, M. Dixon, P. Koutsakis, "An analysis of changing enterprise network traffic characteristics", IEEE Asia-Pacific Conference on Communication (APCC), 2014.
[2] "Transport Services (TAPS)", https://bit.ly/2nLN46u, accessed 2020-01-24.
[3] M. Handley, "Why the Internet only just works", BT Technology Journal, Vol. 24, No 3, July 2006.
[4] R. Stewart et al., "Stream Control Transmission Protocol", RFC 2960, Internet Engineering Task Force, October 2000.
[5] E. Kohler, M. Handley, S. Floyd, "Datagram Congestion Control Protocol (DCCP)", RFC 4340, Internet Engineering Task Force, March 2006.
[6] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, Internet Engineering Task Force, January 2013.
[7] J. Roskind, "QUIC: Multiplexed stream transport over UDP", Google working design document, 2013.
[8] L.-A. Larzon et al., "The Lightweight User Datagram Protocol (UDP-Lite)", IETF RFC 3828.
[9] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A transport protocol for real-time applications", IETF RFC 3550.
[10] "SPDY: An experimental protocol for a faster web", http://bit.ly/2Sh14Az, accessed 2020-01-29.
[11] V. Singh, S. Ahsan, and J. Ott, "MPRTP: Multipath considerations for real-time media", 4th ACM Multimedia Syst. Conf. (MMSys), Oslo, Norway, 2013.
[12] J. C. Mogul, "TCP Offload Is a Dumb Idea Whose Time Has Come", In HotOS, pages 25–30, 2003.
[13] "Data Plane Development Kit", https://www.dpdk.org/, accessed 2020-01-24.
[14] L. Rizzo, "netmap: A novel framework for fast packet I/O", In Proc. of USENIX Annual Technical Conference, pages 101–112, 2012.
[15] N. Khademi et al., "NEAT: A Platform- and Protocol-Independent Internet Transport API", IEEE Com. Magazine, June 2017.
[16] E. Dubois, et al., "Enhancing TCP based communications in mobile satellite scenarios: TCP PEPs issues and solutions", Proc. of the 5th ASMS and 11th SPSC, September 2010.
[17] M.Honda, F.Huici, C.Raiciu, J.Araujo, and L.Rizzo, "Rekindling Network Protocol Innovation with User-level Stacks", ACM SIGCOMM CCR, Apr. 2014.
[18] M. Alizadeh, et al., "DCTCP:Efficient packet transport for the commoditized data center", In SIGCOMM, 2010.
[19] V. H. Tran and O. Bonaventure, "Beyond socket options: making the Linux TCP stack truly extensible", IFIP Networking, 2019.
[20] "Linux man page", http://bit.ly/2O8UGKN, accessed 2020-01-29.
[21] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. "How hard can it be? designing and implementing a deployable multipath TCP", In USENIX NSDI, 2012.
[22] Q. De Coninck et al., "Pluginizing QUIC", ACM SIGCOMM, Beijing, China, 2019.
[23] C. Nicutar, C. Paasch, M. Bagnulo, and C. Raiciu, "Evolving the Internet with connection acrobatics", in Proc. Workshop Hot Topics Middleboxes Netw. Funct. Virtualization (HotMiddlebox), Santa Barbara, CA, USA, 2013.
[24] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska, "Implementing network protocols at user level", IEEE/ACM ToN, 1993.
[25] K. Yasukata et al., "StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs", In Proc. of the 2016 USENIX ATC.
[26] E. Jeong, et al., "mTCP: a highly scalable user-level TCP stack for multicore systems", In Proc. 11th USENIX NSDI, Apr. 2014.
[27] M. F. Nowlan, N. Tiwari, J. Iyengar, S. O. Aminy, and B. Ford, "Fitting square pegs through round pipes: Unordered delivery wire-compatible with TCP and TLS", in Proc. USENIX NSDI, San Jose, CA, USA, 2012.
[28] G. Papastergiou, et al.,, "De-ossifying the Internet transport layer: A survey and future perspectives", IEEE Communications Surveys & Tutorials, vol. 19, no. 1, 2017.
[29] Korian Edeline and Benoit Donnet, "A Bottom-Up Investigation of the Transport-Layer Ossification", In: Network Traffic Measurement and Analysis (TMA) Conference, 2019.
[30] M. Fleming, "A thorough introduction to eBPF", https://bit.ly/2osxU6l, accessed 2019-05-08.
[31] T. Høiland-Jørgensen and al., "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel", 14th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT '18), Dec 2018.
[32] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr., "Exokernel: an

operating system architecture for application-level resource management", In Proc SOSP '95, December 1995.
[33] "netem", https://bit.ly/2owJ36m, accessed 2019-09-22.
[34] H. Hartig, M. Hohmuth, J. Liedtke, S. Schonberg, and J. Wolter. "The performance of $\mu$-kernel-based systems", In Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 1997.
[35] Adam Belay, et al., "IX: A protected dataplane operating system for high throughput and low latency", In Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation, 2014.
[36] M. Zitterbart, "A multiprocessor architecture for high speed network interconnections", in INFOCOM'89. Proc. of the 8th Annual Joint Conference of the IEEE Computer and Communications Societies, vol. 1, Apr. 1989.
[37] H. Kanadia and D. R. Cheriton, "The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors", in ACM symposium on Communications architectures and protocols, 1988.
[38] "NEAT project", http://bit.ly/2SqfTBd, accessed 2019-05-08.
[39] Magnus Karlsson and Björn Töpel, "The Path to DPDK Speeds for AF_XDP", In Linux Plumbers Conference 2018 Networking Track.
[40] Simon Peter, et al., "Arrakis: The operating system is the control plane", In Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14), 2014.
[41] D. Freimuth, E. Hu, J. LaVoie, R. Mraz, E. Nahum, P. Pradhan, and J. Tracey, "Server Network Scalability and TCP Offload", In USENIX'05, 2005.
[42] Mu He et al., "Flexibility in Softwarized Networks: Classifications and Research Challenges.", IEEE Communications Surveys & Tutorials, Vol 21, No. 3, Third Quarter 2019.
[43] M. Fleming, "A thorough introduction to eBPF", https://lwn.net/Articles/740157/, accessed 2020-05-31.
[44] S. McCanne, and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture", USENIX winter, Vol 93, 1993.