VTL : A Stable Framework for Conception, Implementation, and Deployment of Internet Communication Protocols

El-Fadel Bonfoh

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Institut National des Sciences Appliquées de Toulouse

---

Présentée et soutenue par

El-Fadel BONFOH

Le 26 janvier 2021

VTL: Une Architecture Stable pour la Conception, l'Implémentation, et le Déploiement de Protocoles de Communication d'Internet

---

Ecole doctorale : **EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse**

Spécialité : **Informatique et Télécommunications**

Unité de recherche :
LAAS - Laboratoire d'Analyse et d'Architecture des Systèmes

Thèse dirigée par
Christophe CHASSOT et Samir MEDJIAH

Jury

M. Ernesto EXPOSITO, Rapporteur
Mme Isabelle GUÉRIN-LASSOUS, Rapporteure
M. Damien MAGONI, Rapporteur
M. Nicolas VAN WAMBEKE, Examinateur
M. Thierry TURLETTI, Examinateur
M. Christophe CHASSOT, Directeur de thèse
M. Samir MEDJIAH, Co-directeur de thèse

En mémoire de mon père,
Balah Bawih

The ***scientist*** builds in order to study;
the ***engineer*** studies in order to build.

Frederick P. BROOKS Jr.

Remerciements

Mes premiers mots de remerciement vont à mes directeurs de thèse, Christophe CHASSOT et Samir MEDJIAH. Je les remercie de m'avoir confié ce projet de recherche, de m'avoir encadré et fourni les moyens nécessaires pour mener à bout ce projet.

Je remercie chacun des membres de mon jury qui malgré les aléas liés à la situation sanitaire ont bien accepté de participer à mon jury de thèse. Je remercie tous mes rapporteurs, Isabelle GUÉRIN-LASSOUS, Damien MAGONI et Ernesto EXPOSITO. Je leur suis reconnaissant du temps qu'ils ont consacré à rapporter cette thèse et de leurs retours enrichissants. Je remercie enfin mes examinateurs Nicolas VAN WAMBEKE et Thierry TURLETTI pour leurs commentaires et nombreuses réflexions perspicaces sur mon travail.

Cette thèse a été le fruit d'un long travail mené au sein du LAAS-CNRS. J'en remercie le directeur, M. Liviu NICU et tout le personnel. J'adresse mes remerciements également à tous les cadres scientifiques de mon équipe de recherche SARA notamment M. Khalil DRIRA, M. Slim ABDELLATIF pour ses conseils réguliers lors de cette thèse et surtout son aide précieuse lors de la préparation de la soutenance de cette thèse. Je remercie également M. Philippe OWEZARSKI avec qui j'ai eu bien d'échanges fructueux.

J'adresse une spéciale à tous mes collègues et amis qui de près ou de loin m'ont apporté leur soutien à leur manière. Une spéciale dans la spéciale pour mon jumeau et compatriote de tous les temps, qui a été d'un soutien sans faille au cours de toutes ces années, merci Bénoît.

Je remercie ma famille d'avoir pu me soutenir de là où ils sont. Je remercie mon parrain Damien, pour ses conseils et encouragements, ainsi que sa femme Lydie qui m'a fortement poussé à rédiger ce manuscrit, c'est fait. À travers eux, c'est à toute l'association Five Hearts que j'adresse mes remerciements. Enfin, je ne remercierai jamais assez ma complice de toujours, celle qui a toujours supporté de vivre sa thèse en même temps que la mienne. Merci pour ton soutien de tous les goûts Nour.

Abstract

The Internet and its evolutions are fundamentally based on the unique TCP/IP model, whose primary protocol of the Transport layer (L4) is the TCP protocol (and somewhat UDP). Despite its well-known limitations, TCP is still widely deployed and used on almost 90% of Internet traffic. Nearly all the literature's propositions to overcome TCP's limitations are *not deployed* in the mainstream operating systems (OS) of the market and/or face *limited use* by the Internet's applications. This situation leads to the *ossification* or *sclerosis* of the Internet Transport layer and is a significant barrier to the introduction of innovation into this layer of the Internet's TCP/IP architecture. Thus, this thesis proposes to address the issue of ossification of the Transport layer and is focused on three main contributions disseminated in the six chapters of this manuscript. First, we propose and implement the architecture of a service-oriented Virtual Transport Layer (VTL) and extend this service-oriented paradigm by providing the architecture with the capability to *dynamically deploy* Transport protocols within the end-systems operating systems (OS). In order to facilitate the use and stimulate the adoption of the proposed architecture, we then provide the approach and mechanisms necessary to allow any TCP-based application to use *transparently* any Transport protocol other than TCP. The transparency refers to the fact that the TCP application does not need to be modified. One thing is to know how to replace TCP in a transparent way for the application, the other thing is to choose the best alternative to TCP. Indeed, depending on network conditions and application needs, it would be better to choose one protocol over another. The optimal choice of the alternative to TCP according to the network context and the TCP application's needs is the subject of this thesis's last contribution.

Keywords: Transport protocols, TCP/IP, Dynamic Deployment, eBPF/XDP, Protocol Selection, Linux OS.

Résumé

L'Internet et ses évolutions technologiques sont fondamentalement basés sur l'unique modèle de communication TCP/IP dont le protocole principal de la couche Transport (L4) est le protocole TCP (et dans une moindre proportion UDP). Malgré ses limites bien connues, TCP reste très largement utilisé sur près de 90% du trafic Internet. La quasi-totalité des propositions de la littérature pour pallier les limites de TCP sont *non déployées* dans les principaux systèmes d'exploitations du marché et/ou font face à une *utilisation limitée* par les applications sur Internet. Cette situation conduit à ce qui est connu sous le nom d'*ossification* ou de *sclérose* de la couche Transport d'Internet et constitue une barrière importante à l'introduction d'innovation dans cette couche de l'architecture TCP/IP d'Internet. Ainsi, cette thèse se propose d'adresser la problématique de l'ossification de la couche Transport et est centrée sur trois principales contributions disséminées dans les six chapitres de ce manuscrit. Dans un premier temps, nous proposons et implémentons l'architecture d'une couche de Transport virtuelle (VTL) orientée-service et étendons ce paradigme de l'orientée-service en dotant l'architecture de capacité de *déploiement dynamique* de protocoles de Transport au sein des systèmes d'exploitation des hôtes d'extrémités. En vue de faciliter l'utilisation et stimuler l'adoption de l'architecture proposée, nous fournissons dans un deuxième temps, l'approche et les mécanismes nécessaires pour permettre à toute application TCP d'utiliser de manière *transparente* tout protocole de Transport autre que TCP. La transparence fait référence au fait que l'application TCP n'a pas besoin d'être modifiée. Une chose est de savoir remplacer TCP de façon transparente pour l'application, l'autre chose est de choisir la meilleure alternative à TCP. En effet, en fonction des conditions du réseau et des besoins de l'application, il serait plus judicieux de choisir tel protocole plutôt que tel autre. Le choix optimal de l'alternative à TCP suivant le contexte réseau et les besoins de l'application TCP est l'objet de la dernière contribution de cette thèse.

Mots-clés: Transport protocols, TCP/IP, Dynamic Deployment, eBPF/XDP, Protocol Selection, Linux OS.

Contents

List of Figures

List of Tables

Introduction 1

1.1 Thesis Big Picture

The last two decades have seen a rapid expansion of the Internet and its place in human life. The recent Covid-19 pandemic proves it: teleworking, teleconference, teleteaching, telehealth, etc., have been massively used. This growth of the Internet has been and remains possible thanks in part to the introduction of a plethora of new paradigms (Cloud/Fog/Edge Computing, SDN, NFV, etc.) and their associated technologies (OpenStack, OpenFlow, eBPF, Docker, etc.). Those paradigms and technologies bring new opportunities and open up new challenges. For instance, Software-Defined Networking (SDN) paradigm [1] and its technical expressions OpenFlow and P4 introduced *programmability in the core* of networks and eased the configuration, maintenance, and monitoring of networks. In the wake of SDN, Network Function Virtualization (NFV) [2] and its relevant implementations enabled more *agility* and *flexibility* in managing and deploying network services. At the same time, more recent technologies such as extended Berkeley Packet Filter (eBPF) [3] allowed to dynamically insert functionalities (protocol functions, packet filters, etc.) in the operating system (OS) kernel of end-systems, hence introducing *programmability at the edge* of networks by temporary modification of end-systems behavior.

This proliferation of new paradigms and technologies forms a *"softwarized"* Internet ecosystem (Fig. 1.1) characterized by a broad *heterogeneity* and high *volatility*. Indeed, they are heterogeneous in terms of opportunities (programmables network and OS, . . .) as well as in terms of network constraints (delay, loss, ...) and application requirements (reliability, security, . . .). Furthermore, during the same data transfer session, the network state is constantly changing due to a number of events such as the migration of resources (VMs, Containers, etc.) between datacenters, hardware and software failures, datacenter servers overloading, or link quality degradation. This leads to a very volatile ecosystem that requires the design of auto-adaptive systems and frameworks able to follow up these dynamic changes in order to ensure network stability and to provide optimal quality of service (QoS) to applications and, at the end, a better quality of experience (QoE) for the users.

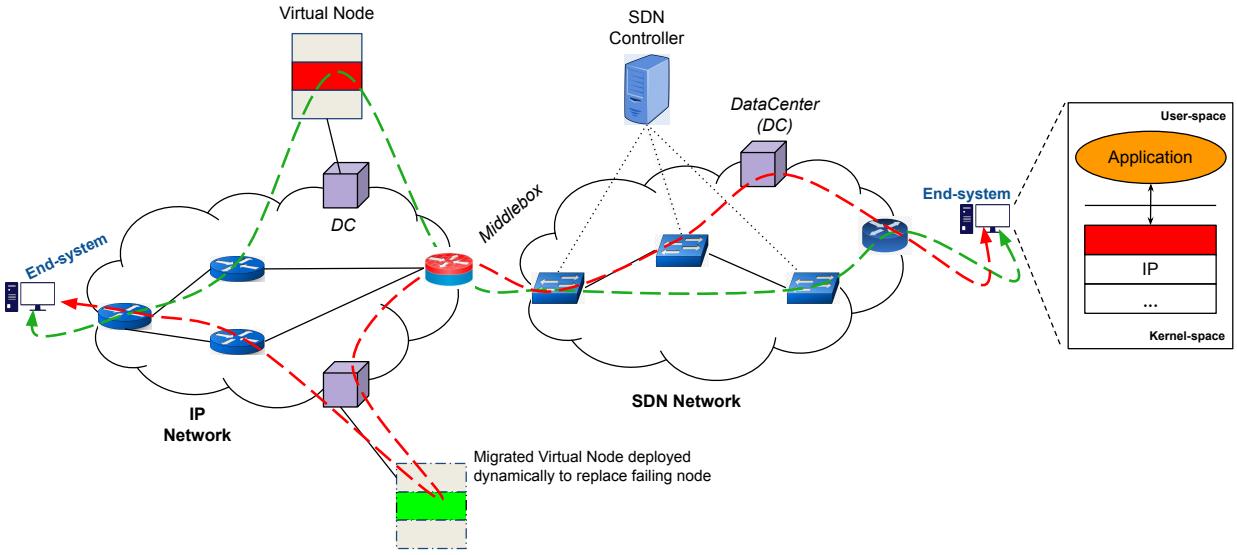


Figure 1.1: Softwarized Network Ecosystem Illustration.

Besides, the Internet is the crossroads of more and more stakeholders (OS providers, applications programmers, middleboxes vendors, etc.), each one arriving with its own requirements and expectations. For instance, while application programmers could significantly improve their applications' performance by using a tailored but not necessarily standardized L4 protocol, middleboxes owners could, for security and reliability reasons, prohibit the use of any non-standardized L4 protocol on their equipment. Those mostly opposed and contradictory requirements and expectations hamper the introduction of innovation and lead to the well-known *ossification* of Internet architecture, especially its communication stack.

1.2 Thesis Scope and Problem

The Internet and its evolutions are essentially based on the same communication model: the TCP/IP standard model, whose main L4 protocol for end-to-end data transfer between network applications remains TCP (and somewhat UDP). Indeed, with the evolution of the Internet, several Transport layer protocol mechanisms have been proposed in the literature to continually improve end-to-end QoS with the aim to satisfy the increasing requirements of applications and to adapt to a variety of emerging networks. Unfortunately, most of these protocols either are *not available* within the mainstream OSes and/or suffer from a *limited use* within the Internet (see Fig. 1.2 that shows the distribution usage of Transport protocols on the Internet). Results (from [4]) reported in Fig. 1.2 (a) show that almost 90% of Internet traffic is based on TCP. Fig. 1.2

(b) and Fig. 1.2 (c) are additional datasets that we collected during a preliminary experiment by capturing incoming and outgoing Internet traffic to and from two Linux servers running different OS. The study under the Wireshark analyzer tool [5] of the collected data provides two main insights during this preliminary experiment. On the one hand, to navigate for instance on youtube.com or google.com, the same application (Chrome) uses different Transport protocols depending on whether it is running under Linux 4.10.0 or under Linux 5.3.5 ⇒ *the used protocol depends on its availability on the OS*. On the other hand, the same kind of application (i.e., web browser, here Chrome and Firefox ESR) surfing on the same website does not use the same Transport protocol ⇒ *the used protocol depends on the one adopted by the application programmers*.

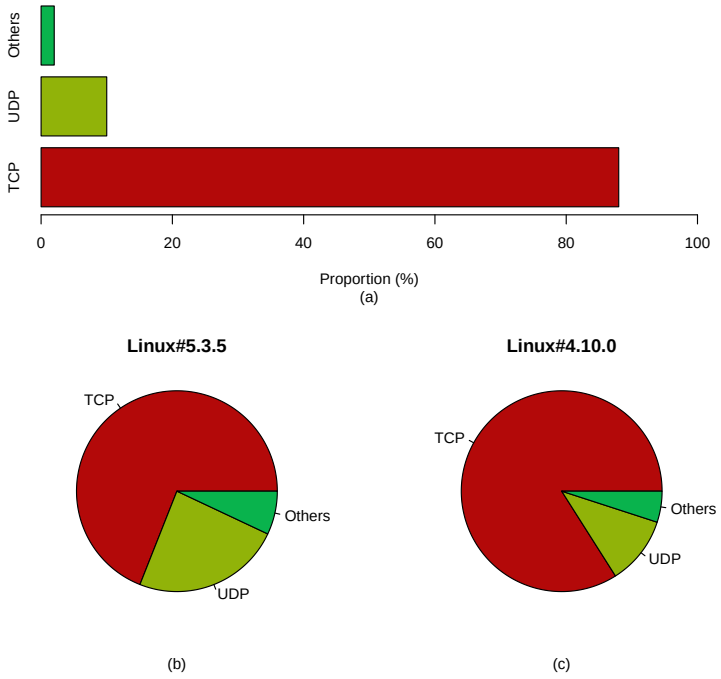


Figure 1.2: Usage Distribution of Transport protocols. (a): Almost 90% of the Internet's applications use TCP. (b) and (c): Depends on the protocol's availability on the operating system (OS) and the programmer's choices, the trend remains similar.

Summary and research questions. The review of the literature and the analysis of the results of preliminary experiments we carried out lead us to the following conclusion: 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* of the Internet Transport layer, also known as *ossification* of the Transport layer. The Internet Transport layer ossification is a major barrier to the introduction of more agility and flexibility enabled by the evolution of new paradigms in the Internet such as the emerging "softwarized" networks and their main underlying technological concepts SDN and NFV. This thesis 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 mechanisms on the end-system, *how, on the one hand, to ensure the deployment of the appropriate protocol to the application requirements and network state and, on the other hand, to ensure its seamless usage by (existing) applications*? The focus is placed on the exploration of opportunities provided to us by in-kernel dynamic deployment technologies, namely eBPF.

1.3 Thesis Contributions

Apart from the extensive evaluations performed during our journey, the main contributions of this thesis are listed below.

Dynamic grafting of Transport protocols. To address the Transport layer's ossification, most of the previous research efforts proposed replacing the *limited* socket API with a *service-oriented API* so that the application no longer invokes a specific protocol but asks for a Transport Service[1] . The choice of the appropriate protocol to provide the required service is delegated to the Transport layer. However, in its current state, this approach is based on the assumption that the required protocol is already available on the OS of the end-system and, therefore, lacks a method to dynamically deploy a new protocol (or protocol component) when it is not available at the end-system. To fill this gap, we extend the service-oriented approach by introducing the concept of *dynamic deployment* of protocol grafts. Dynamic grafting of protocols consists of *on-the-fly* integration of protocol components within the end-system either at the *user-space* or the *kernel-space* of the operating system (OS). Dynamic and on-the-fly properties refer to the fact that we provide the technique ensuring that the deployments are realized without the need to recompile and reboot the OS.

1: A *Transport Service* is an abstraction of a set of Transport functions that provides an end-to-end facility to applications. Examples include but are not limited to: *reliable* delivery service, *ordered* delivery service, *partially ordered* delivery service, *encrypted* delivery service, etc.

A *protocol graft* is a pluggable protocol component that consists of a set of Transport functions with their associated interaction pattern. The interaction model defines the way the Transport Functions are orchestrated. Chapter 3 describes in more details this notion.

Transparent reconfiguration/integration of legacy applications. As previously stated, our first contribution is an extension of the service-oriented approach that is under standardization within the IETF working group TAPS [6]. However, once the standardization stage is over, we believe a major problem will arise: porting current legacy applications[2] to the new service-oriented API. This will require a modification of the applications that could be a barrier to the adoption of this approach, as we will see in the subsequent chapter. This might result in *limited use* of *(1)* the protocols integrated within the service-oriented architecture as well as *(2)* the architecture itself. To prevent this eventual limited adoption, we design and implement an approach that permits to replace at runtime TCP by another protocol *X*. We realize it in a *transparent* way to TCP applications, i.e., there is no need to rewrite the code of the latter. During the experiments of the proposed approach, we

2: Throughout this manuscript, legacy application is defined as TCP application that uses the standard socket API to consume Transport layer services. We will use interchangeably the terms "legacy application" and "TCP application".

also notice that the suitable Transport protocol *X* that should be used as an alternative to TCP varies depending on the application requirements and the network conditions. The goal of our last contribution, listed below, is to enable the selection of the most appropriate L4 protocol to replace TCP in a given context.

Optimal selection of protocols. As stated above, we noticed that if the choice of the protocol *X* is made blindly, the application could present suboptimal performances (often) lower than its initial performances under TCP. Hence, we propose an approach that must ensure the selection of the best alternative to replace TCP. This choice is driven by a set of machine learning models, namely decision trees that we trained to feed the knowledge base of our decision algorithms. The attributes of the decision trees are the applications requirements and the network conditions. Therefore, prior to the selection of the better L4 protocol, we propose *(1)* a profiling method that allows inferring the requirements of the (legacy) application and *(2)* a parsimonious monitoring that is useful to estimate the state of the network in terms of RTT, loss rate and maximum available bandwidth.

We realize the above contributions within Virtual Transport Layer (VTL), a protocol deployment and data delivery management system designed and implemented during this thesis. VTL follows three main design principles: *1/* the ***seamless support of legacy applications***, i.e., legacy applications might consume Transport layer 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, 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 must be organized in such a way to allow the implementation of reconfigurable protocols whose components might be dynamically instantiated and parameterized. We implemented VTL by leveraging and combining two kernel subsystems: `XDP` and `TC`, part of eBPF technology. VTL design principles and implementation tools are described more in-depth in the next chapters.

Throughout this manuscript, *aware-application* is defined as a new brand of application that uses the API provided by VTL system to consume Transport layer services. See Chapter 3 for more details.

At last, although we fully address Transport layer sclerosis at the *end-systems*, more investigation and algorithms are required to deal with the middleboxes that populate the *core of* the Internet. In fact, the current algorithm integrated into our approach seems limited and consists of systematic fallback to TCP or UDP in case of rejection by one middlebox.

1.4 Dissertation Structure

Except for the conclusion, subject of ***Chapter 6***, the rest of this dissertation is structured around four main chapters.

Chapter 2, after an insightful analysis of the Transport layer ossification causes, revisits the limitations of previous works that address the ossification of the Transport layer. The lessons learned from this analysis allow us to lay down the fundamentals for what could be the requirements and design principles for VTL. We conclude the chapter by presenting these design principles.

Chapter 3 first introduces VTL and presents its key concepts. Then, we present the detailed functional architecture of VTL by emphasizing on the aspects of dynamic deployment of protocols as well as on the interaction with *aware*-applications. Finally, we conduct extensive experiments to evaluate VTL, namely the deployment delay of the protocols and the performances (in terms of data transfer rate and latency) of those protocols under VTL.

In ***Chapter 4***, we present our approach that allows legacy applications to *transparently* replace TCP with another L4 protocol at *runtime* (i.e., during the application execution). Then, we discuss the implementation and evaluate the "costs" and benefits induced by our proposed approach. Finally, we conclude the chapter by analyzing the evaluation results and discussing the findings that motivate the last contribution of this thesis.

Finally, ***Chapter 5*** presents the last contribution of this thesis. It details the approach we adopted to achieve the selection of a better alternative to TCP in order to maximize the performance gains of legacy applications. To conclude the chapter, we evaluate our proposed selection algorithm's precision, and we estimate its benefits, i.e., the average performance (in terms of throughput) it permits the legacy applications to gain.

Background: Related Work and Thesis Positioning

2

This chapter aims to introduce the general approach followed during this thesis and position it regarding the previous works. To this end, the chapter is organized into four parts as follows.

First, we provide an insightful literature review that enumerates the different elements that hamper the introduction of innovation within the Internet's Transport layer. Those elements could be summarized by the so-called vicious circle, which is introduced in Section 2.1. Then, Section 2.2 presents the previous research efforts that address the Transport layer's ossification issues and points out the shortcomings of those works. Learning from this preliminary review of the literature, we introduce in Section 2.3 the requirements and design principles of a new Transport layer system called VTL, which aims to enable the deployment of any new Transport protocol (or protocol component), and ease/stimulate its use by legacy as well as aware-applications. Finally, in Section 2.4, we conclude the chapter.

2.1 Vicious Circle

The deployment and wide adoption of any Transport protocol mechanisms on the Internet rise up several challenges and require taking into account three main stakeholders:

- The run-time environment of the protocol components principally managed by *OS developers*,
- The consumers of the services provided by the protocols, namely *application programmers*,
- The *middleboxes vendors* which maintain the network infrastructure over which data packets handled by the protocols are transmitted.

Each one of these actors comes with its own requirements and expectations. Thus, the foremost complexity is to find approaches that consistently meet the often opposed requirements of these actors.

2.1.1 Deployment Barriers

The Operating System (OS) constitutes the location environment of the Transport protocols. Basically, every *normal* process (i.e., user application) running on the OS has a virtual memory associated with it. As illustrated in Fig. 2.1, OS separates the *virtual memory* space into two main parts: the *kernel-space* and the *user-space*[1] . The user process owns the latter whereas the kernel-space is common to all processes running on the system. Therefore, to prevent user processes from interfering with each other, the OS uses a security model based on *system calls* (syscall for short) to control access to the kernel-space. System calls allow a user process to interact in a safe fashion with the kernel and to delegate to it critical tasks that take place in the kernel-space such as accessing a network device in order to manipulate raw data packets, creating a new file, changing files' permission, etc. As soon as a user process issues a syscall, the processor changes its privilege level[2] and switches from *user mode* to *kernel mode*. Then the kernel checks the permit of the requested task and performs it on behalf of the user process.

1: The term *userland* is often employed

2: Processors offer several privilege levels that define the permitted actions of the process. Depending on the attributed privilege level, the process might execute certain assembly language instructions, access to specific parts of the virtual address space, etc. In Linux OS, processors use two different modes: kernel mode and user mode. The major difference between the two is that access to the kernel-space of the virtual memory area is forbidden in user mode.

System calls guarantee the safety of the system. However, due to their built-in operations (functions checks, contexts switching, etc.), the path from the user program to the network interface card (NIC) during data transfer could be very slow. Thus, for self-evident reasons of performance, early works on the Transport layer suggested implementing and executing the protocol mechanisms in the kernel-space of the OS [7, 8]. Most of them went further and proposed to offload these mechanisms in the NIC driver of the end-system [9–12]. However, at the expense of performance, a series of studies [13–16] proposed to move protocol stacks in the user-space 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

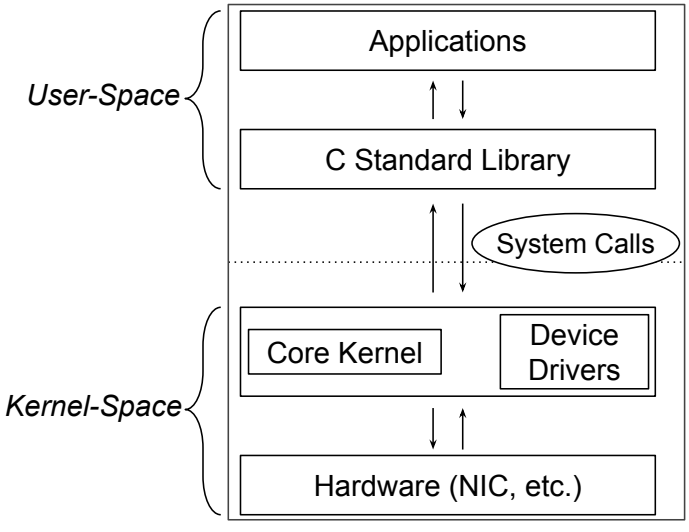


Figure 2.1: High-level overview of general-purpose Operating System (OS) structure.

as DPDK [17] or NetMap [18] to reduce performance degradation induced by the execution of protocols within a user-space of the OS. Nevertheless, user-space protocol implementations leave security and efficiency concerns on the table (see Section 2.2).

Altogether, the appearance of software acceleration tools such as DPDK or NetMap is *relatively* recent. So, as stated in the above paragraph, the choice that has 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 an upgrading of their system. OS upgrade is not only time-consuming, very tedious, and error-prone, but it also poses significant software and hardware compatibility issues. For instance, let us consider a memory error such as an attempt of access to an unavailable or not-reserved memory address. In the user-space, this error is managed by the kernel and will lead to a "simple" *segmentation fault* (or *core dump*) and the halt of the application involved in the error. However, the same error in the kernel itself might take down the whole OS "at best" as soon as the error occurs, and at worst, several hours after the appearance of the error, which further complicates debugging that is basically difficult due to a dire lack of kernel-friendly debugger tools. Any modification of the kernel code requires the utmost attention to detail. Those modifications are therefore left to the experienced developers of OS vendors and even for open systems like Linux, it takes *benevolent dictators*[3] 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 and quick-off the integration of any new protocol solution.

3: The term "benevolent dictator" was used by Eric Steven Raymond to designate, in a project, the developer who decides whether improvements should be integrated or not into the system. The latter usually intervenes to settle disputes and to ensure the sustainability of the system.

2.1.2 Limited Adoption Root Cause

The standard model TCP/IP of the Internet is a layered model in which each layer offers services to the upper layer and leverages the services provided by the lower layer. Generally, each layer exposed its services via standard interfaces containing the signatures of functions that implement the exposed services.

The main interface used by applications to consume the services provided by the Transport layer (protocols) is the standard *socket API* [19]. This API, presented later more in-depth in Chapter 4, is designed in such a way that the application programmers are required to *explicitly* choose the L4 protocol at the design-time of the application (i.e., when the code is written). The first evident consequence is that the Transport services (provided to the applications) are limited to those of the chosen Transport

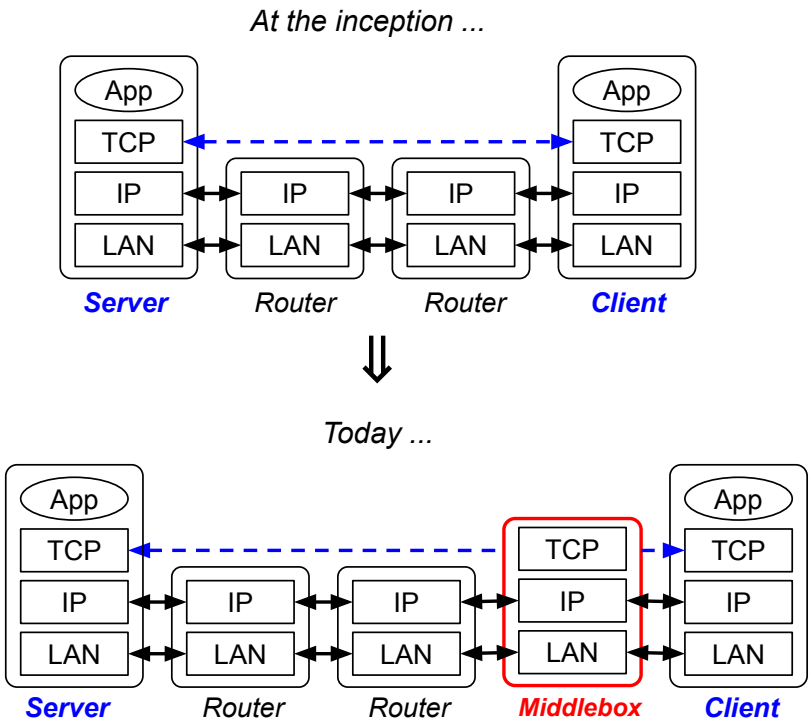


Figure 2.2: The impact of middlebox on the end-to-end principle.

protocol. Also and foremost, to adopt any new protocol solution, programmers must modify their applications' code. This latter corollary might be a factor of more and more 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 needs of the application.

Our analysis is that to prevent the issues associated with frequent modifications and ensure the stability of their applications, most of the application programmers prefer to rely on standard protocol solutions such as TCP or UDP, which are recognized as stable and available on the mainstream operating systems 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 the requirements of their applications.

2.1.3 Other Factors Out of End-systems

At the birth of the Internet, the logic that prevailed was based on the *end-to-end principle* [20] where the network, composed mainly of routers, had no visibility over what happens beyond the L3 level. The only concern of any network router was to forward L3 packets to the next hop towards the final destination by looking exclusively at the IP header information of the packets. The end-systems should implement the L4 and above additional services such as reliability, security, etc. The end-to-end principle clearly defines and separates the role of each networked component. Its goal was

to enable the introduction of new applications and services at the edge without the need to modify the core network, i.e., the routers.

For a long time, this principle was the subject of global agreement and ensured the popularity and stability of the Internet's TCP/IP architecture. Nevertheless, the situation has changed today, especially for network operators. The latter need to have visibility on the flow of data packets passing through their equipment in order, for example, to troubleshoot and repair failures or to apply judicious differentiation rules on the services they provide. These requirements have gradually led them to populate the network with *middleboxes*. A middlebox (illustrated in Fig. 2.2) is a kind of enhanced router able to read and modify packets up to the L4 level and decide to reject any unrecognized protocol. Examples include but are not limited to NATs, IPS/IDS, firewalls, or proxies. The massive introduction of middleboxes has "violated" the end-to-end principle but permits network operators to meet their requirements in terms of performance improvement, security enhancement, optimization of resource usage, fast network troubleshooting, etc. To ensure all of these requirements, middleboxes vendors (and somewhat network operators) are often unwilling to configure their devices to whitelist any new protocol until the most popular OS developers support this latter protocol.

All in all, this global context leads to the well-known vicious circle that we illustrated in Fig. 2.3. 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.

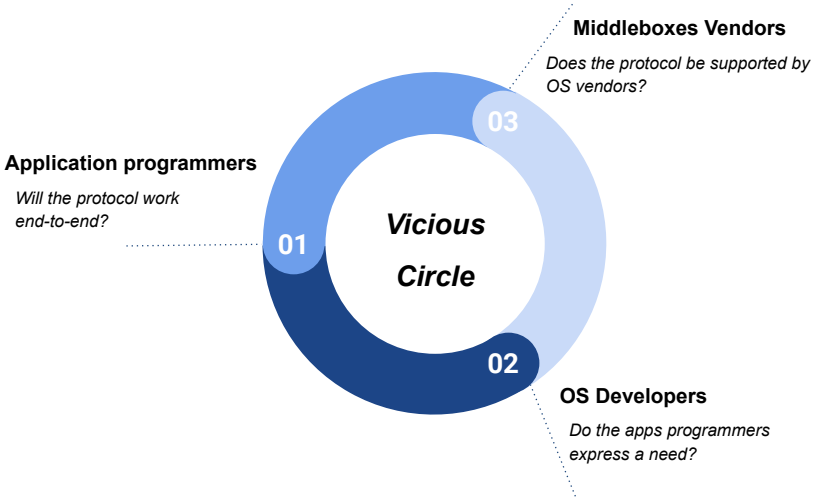


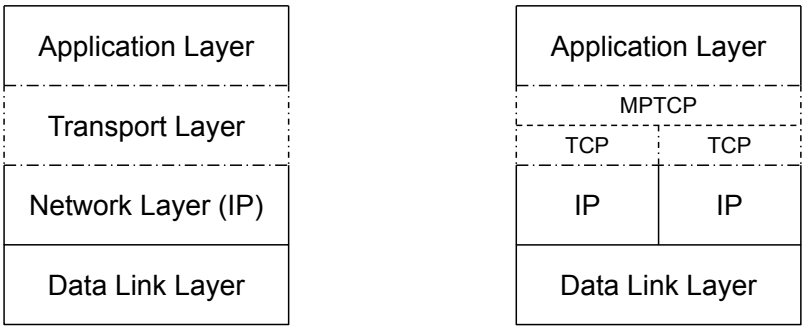
Figure 2.3: Internet's stakeholders forming the vicious circle.

2.2 Transport Layer: Protocols and Architectures

In the standard 4-layer TCP/IP model illustrated in Fig. 2.4 (a), the Transport layer occupies a pivotal position between the high and low layers. Fundamentally, the Internet's Transport layer ensures end-to-end reliable data moving between applications and provides multiplexing services based on the port numbers. Additionally, it ensures the regulation of data transmission at end-systems as well as the collapse of the network by preventing/reducing the congestion and its side effects. As already stated, its traditional protocols are TCP and UDP. The services provided by TCP and UDP quite reflect the primary services and characteristics of the Internet's Transport layer. TCP is a *connection-oriented* and *byte-stream-oriented* protocol that offers full reliable and total order Transport services to applications. In contrast, UDP is a *connectionless* and *message-stream-oriented* protocol that offers minimal checksumming and multiplexing Transport services. UDP does guarantee neither reliability nor order of the data packets delivered to the application.

Connection-oriented vs. Connectionless. A typical Transport session usually occurs in three stages of operation: connection establishment, data transfer, and connection teardown. A connection-oriented Transport protocol provides mechanisms for each of the three operations. Moreover, a connection-oriented protocol maintains state information about the connection (e.g., data packets sequence number, congestion window size, maximum packet size (MSS), etc.) during the whole session. If no state information is maintained and if the Transport session only supports the data transfer stage, the Transport protocol is connectionless.

Message-stream-oriented vs. Byte-stream-oriented. When the application submits its payload to the Transport layer — let us assume the message is 1KB size — the L4 protocol could segment or not the message before encapsulating it and sending it through the IP layer. If the L4 protocol segments the application data (for instance, in two 0.5KB messages), the protocol is byte-stream-oriented. The boundaries of the original data of the application are not preserved. Each of the 0.5KB messages has no significance for the application; it is just raw bytes: a reassembly mechanism is mandatory at the receiver side. In contrast, when the protocol is message-stream-oriented, it preserves the application data's boundaries: the 1K message is delivered as it is without any segmentation from the sender to the receiver. If required, the segmentation should be performed at the underlying IP layer.



(a) Standard TCP/IP Model (b) MPTCP Protocol Stack

Figure 2.4: Internet TCP/IP layered architecture and MPTCP overview.

As the Internet grows, TCP and UDP services appear more and more too limited to meet new (notably *multimedia*) applications' requirements and emergent networks' characteristics. To enrich the Internet's Transport layer with additional Transport services as well as bring to it more *flexibility* and *extensibility* , many research propositions have emerged [21, 22]. Roughly, those works take two research directions: *(1)* proposing a *single/specific protocol* and *(2)* rethinking the *whole Transport layer* architecture in order to arrive at a stable, configurable, and extensible communication framework. We discuss some of those researches in the below sections (see Table 2.1). The following presentation does not aim to describe the internal functioning of the discussed protocols and architectures but, for each of the discussed solutions, we focus on

By flexibility, we mean the degree of ease and time required for a protocol to be modified in its internal architecture, updated and redeployed in its production environment. The extensibility of the protocol designates its ability to allow the add of new functionalities to it.

- *(i)* the added values of the solution in terms of the new services it enriches the Transport layer with,
- *(ii)* the solution's technical specifications (tools and implementation spaces),
- *(iii)* and the solution's interactions with the various elements of its external environment, in particular applications.

The final goal is to learn from the limitations of these solutions regarding the ossification of the Internet's Transport layer.

2.2.1 Innovations' Enforcement at End-systems

Recall there are two spaces of the operating system (OS) that can accommodate the Transport protocols: user-space and kernel-space. To "force" the integration of new protocols within the OS, three approaches exist: *(1)* act like an OS developer and implement in kernel-space the extensions to standard protocols such as TCP, *(2)* implement a part of the protocol in user-space on top of UDP, *(3)* or move the entire protocol into user-space by leveraging software acceleration tools.

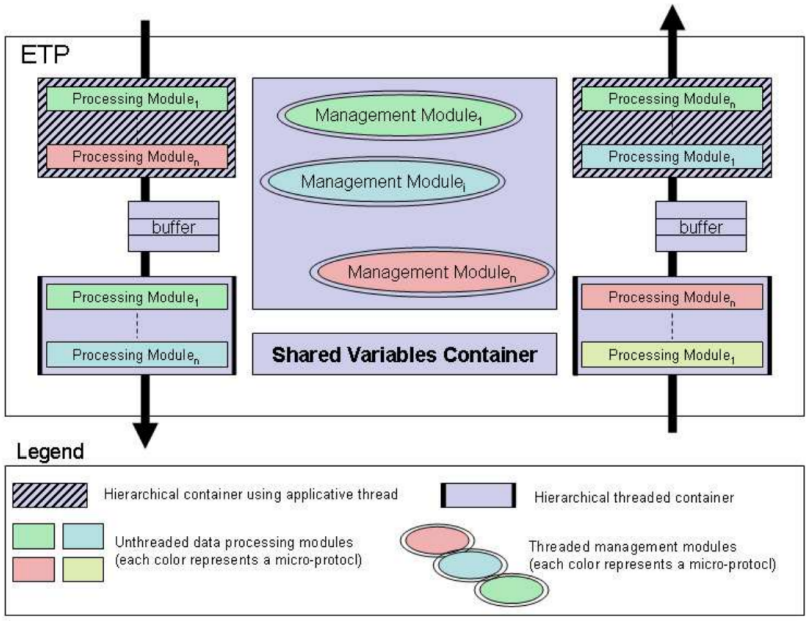


Figure 2.5: High-level view of ETP architecture.

(1) Standard Protocol Extensions and OS Kernel Patches. A kernel patch is a (huge) piece of code integrated into the kernel and that repairs the latter or adds to it new features. Extending standard protocols such as TCP and implementing those extensions as a set of kernel patches [23] is a very common approach to add new services to the Internet's Transport layer. The most recent and famous example of this approach is the Multipath TCP (MPTCP) protocol [24]. MPTCP is a set of extensions that add a multipath capability to the regular TCP. It enables a TCP host to simultaneously use multiple network interfaces (4G/5G, Wi-Fi, etc.). This permits bandwidth aggregation and ensures the resilience of the Transport connection. The architectural overview of MPTCP is provided in Fig. 2.4 (b).

(2) UDP and User-space Protocols Libraries. Operating system (OS) kernel modification is tricky (see Section 2.1); 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. Two protocols (among others) perfectly illustrate this approach: the modular ETP [25] and the emerging QUIC [26, 27].

ETP, an enhanced version of FPTP framework [28, 29], is a modular and adaptive Transport protocol that primary purpose is to equip and reinforce the Transport layer with dynamic *behavioral* and *structural* adaptation properties. Behavioral adaptation consists of keeping the same Transport protocol and tuning its parameters in order to modify the Transport services provided to the application. For instance, congestion window resizing is a sort of behavioral adaptation. In contrast, structural (or architectural) adaptation is achieved by an integral replacement of the Transport mechanisms

with other ones more adapted to the ongoing network state to meet the application requirements. Replacing the congestion control (e.g., DCCP TCP-like [30] by DCCP TFRC [31]) is an example of structural adaptation. Fig. 2.5 illustrates the high-level view of ETP architecture. ETP is implemented in the userland as a Java library and relies on UDP to send and to receive data throughout the network.

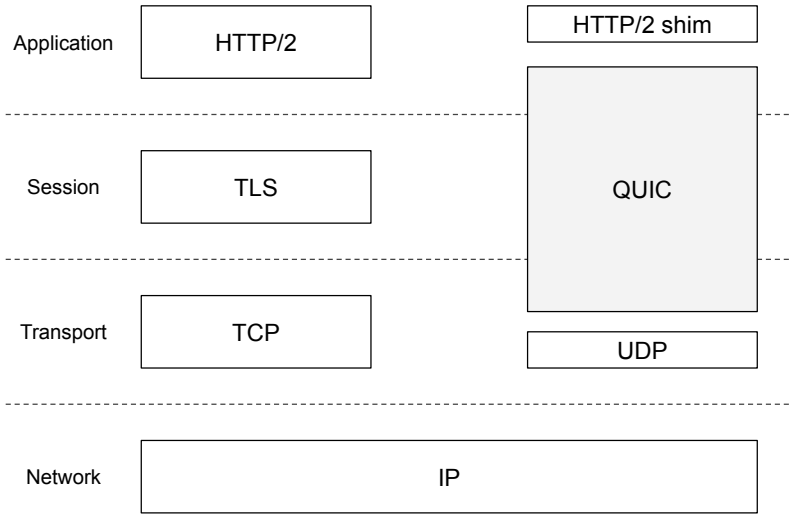


Figure 2.6: QUIC overview in the Internet protocol stack.

Originated at Google in 2012 and nowadays under standardization at IETF, QUIC is a low-latency and security-oriented Transport protocol. As such, in addition to the traditional Transport services, i.e., reliability and order, QUIC key features are a *built-in* (payload) encryption service and a low-latency (connection establishment) service. There are more than twenty official QUIC's implementations [32] developed in almost all standard languages: C/C++, Go, Java, Rust, etc. All of these implementations, as in ETP, are userland libraries that depend upon UDP to interact with the network. Fig. 2.6 shows QUIC protocol position in the layered Internet TCP/IP architecture.

(3) Speedy Full User-space Protocols. Deploying a protocol in the user-space above UDP facilitates the protocol extension and increases its update/upgrade frequency. However, performances would take a hit for two reasons. First, the use of UDP at the underlying level implies that the data packets pass through the kernel network stack which will generate overheads due to several additional operations (not necessarily useful for the protocol itself) imposed per packets: *generic* socket buffer allocations, multiples memory copies, system calls, scheduling, IP table handling, etc. Second, the protocol mechanisms already provided by default by the OS kernel should be reimplemented in user-space by the protocol. For example, QUIC libraries have one or more embedded congestion control mechanisms. Recall that a user process is slower than a kernel thread (see Section 2.1). As a result, the congestion

controls (in general, the protocol mechanisms) implemented in the user-space have limited performance compared to the congestion controls built-in in the kernel with TCP [7, 33]. To eliminate unnecessary overheads generated by the OS kernel and 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, NetMap, or PF_RING [34].

Typically, a kernel bypass tool (a.k.a. software acceleration toolkit) removes the needless kernel operations (enumerated above) and provides to userland programs *shared memory buffers* where the latter could directly access and get data packets as well as put their payload for immediate transmission over the network interface driver (NIC). These shared memory buffers are keystone for the so-called *zero-copy* data transfer technique that reduces memory copies costs and enables the data processing acceleration. Hence, the L4 protocol totally moved in the user-space above DPDK or NetMap should experience better performance. UTCP [13] 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 using kernel bypass tools might still pose some security concerns. Indeed, without any proper control mechanisms such as those provided by the OS kernel (namely system calls), the shared memory between userland and kernel could permit the user applications to access critical kernel memory areas without any precaution and result in the panic and crash of the system at the slightest mistake.

In summary, each of the three above approaches[4] demonstrates how to "force" the deployment of new protocol mechanisms 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 2.1.

4: *(i)* Standard Protocol Extensions and OS Kernel Patches, *(ii)* UDP and User-space Protocols Libraries, and *(iii)* Speedy Full User-space Protocols.

2.2.2 Rethinking the L4 layer architecture and its interactions with Applications

Instead of proposing a single protocol, more recent research investigations to address Transport layer sclerosis propose to rethink the whole architecture of the Transport layer in a way to eliminate current socket API limitations [35]. Their main idea consists of replacing the regular 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 Transport protocol at its design-time but

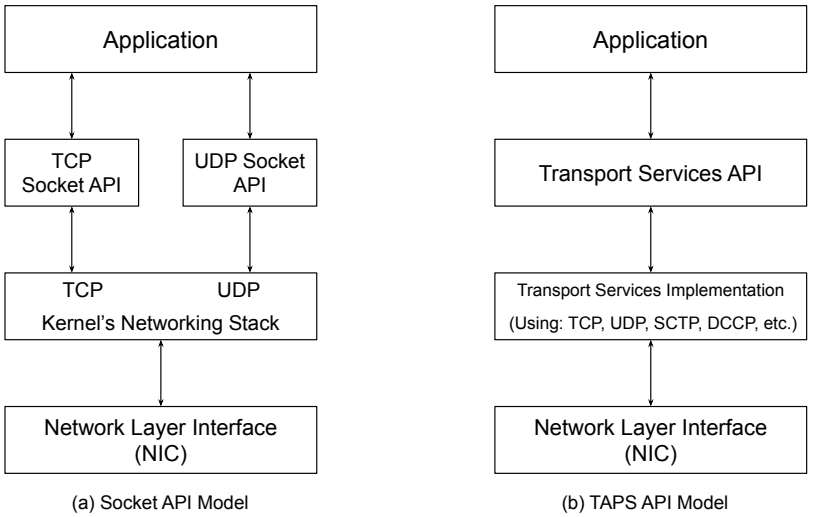


Figure 2.7: Socket API model (a) vs. Transport Services (TAPS) API model (b).

should rather specify the Transport services it wants. The mapping between the required Transport services and the L4 protocol that matches those services is done at runtime (i.e. when the application is executed) by the Transport layer system itself. This mapping depends on the available Transport protocols as well as on the network conditions. Such an approach aims to ease the adoption of all protocols available on the end-system by non-legacy applications. The Transport layer architecture proposed by Mohamed Oulmahdi during his Ph.D. thesis [36] is in line with the aforementioned service-oriented approach. In 2014, an IETF working group named TAPS [6] was chartered to promote and lead this approach's standardization efforts. As of the writing of this thesis, NEAT is the single official implementation of the TAPS standard.

Transport Services (TAPS) standard. TAPS working group aims to: *(1)* define the subset of Transport services that are common to the existing (IETF) Transport protocols services, and *(2)* expose the identified Transport services through an abstract service-oriented API. Further, TAPS specifies the necessary procedures to discover and to select the appropriate protocol that meets the required Transport services. The currently proposed TAPS API model is illustrated and compared to the standard socket API in Fig. 2.7.

NEAT framework. NEAT [37] is an integral implementation of the TAPS standard that exposes to applications a TAPS-like API and allows the latter to express their desired/required Transport services without the need to specify the Transport protocol to use at their design-time. NEAT is implemented in C language as a user-space library above most common (IETF) Transport protocols (TCP, UDP, SCTP, etc.) running either in the user-space or in the kernel-space. Fig. 2.8 shows the architecture of the NEAT system.

Despite its potential to simplify the way applications consume Transport layer services and to break the dependency (especially

the static binding) of the application to a single protocol, the service-oriented approach presents two major drawbacks. First, it relies on the assumption that the best Transport protocol to use 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; the choice of the appropriate protocol is then dependent 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 for the reasons mentioned in Section 2.1.

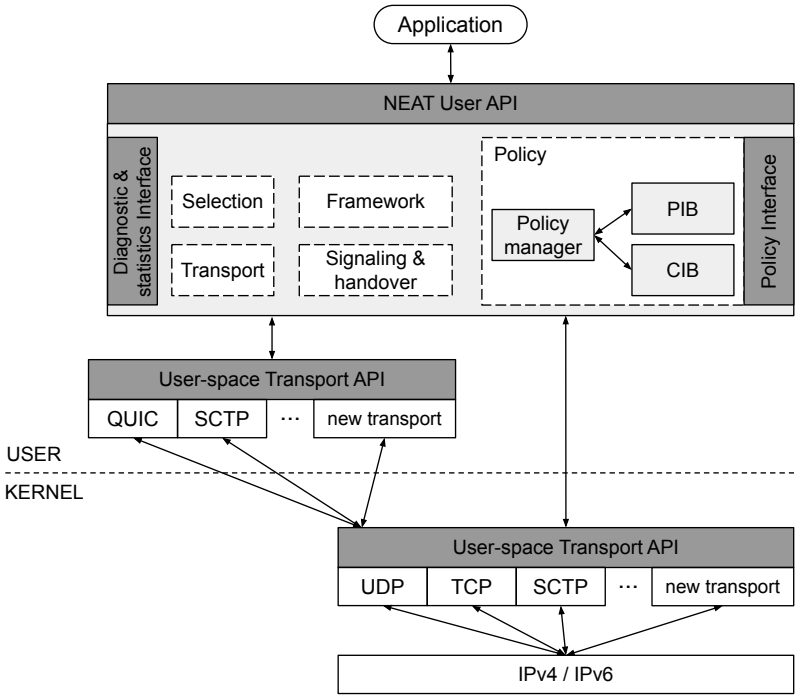


Figure 2.8: The architecture of the NEAT framework [37].

2.2.3 Handle Middleboxes Traversals

In addition to the issues related to the protocols' integration within end-systems, more recent Transport layer solutions incorporate the fact that the protocols, to be effectively operable anywhere on the Internet, must take into account the middleboxes present at any point of the Internet. Those middleboxes' operations may have an undesired effect on the protocols functioning. As mentioned in Section 2.1, middleboxes can reject the packets whose format, i.e. protocol, is unrecognized. They can also modify the packets even if the protocol is allowed on the middleboxes. This is what Network Address Translators (NATs) do. To achieve their primary task i.e. the translation between public and private addresses, NATs need to modify the packets passing through them (see Fig. 2.9). Without necessarily being designed to, NATs operations could

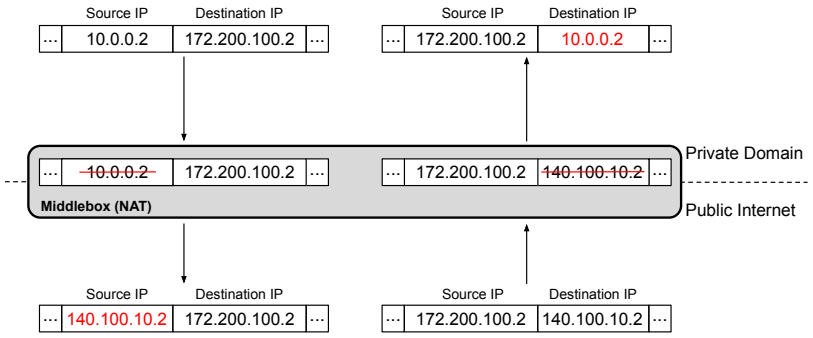


Figure 2.9: Illustration of NAT middlebox interactions on data packets.

effectively impact the well-functioning of the Transport protocols. For instance, MPTCP wide-deployment has been hampered by NATs' presence on the Internet [38]. Additionally to NATs, the Internet is plenty of several examples of middleboxes, namely *(1) firewalls* that based on the packets headers information could drop or modify data packets, *(2) "specialized" middleboxes* dedicated to the modification of Transport headers information [24, 39, 40], (3) etc.

Three main techniques can eliminate (at least limit) the impact of middleboxes on the protocol: the *signaling*, the *dissimulation*, or the *fallback* to TCP or UDP. Let us consider a NAT middlebox to illustrate each of these techniques.

Signaling vs. Dissimulation

Signaling. Signaling consists of initiating, before the data transfer, a negotiation between both end-systems and the various elements of the network to agree on the various parameters (e.g., the L4 protocol) to use throughout the data transfer. If successful, this negotiation allows the application of special treatment to the packets of the *flow* concerned by the negotiation. Let us take the most common example from the literature and call it "example 1". In example 1, we consider two Skype clients, A and B, trying to connect via the Internet. Two recurrent cases are possible: either only one of the two clients (say client A) is behind a NAT, or both clients are behind a NAT. NAT works as follows: by default, it allows only outgoing sessions to traverse it; incoming packets are dropped unless the NAT identifies them as being part of an existing session initiated from within the private network. This mode of operation corresponds to the functioning of the so-called *traditional* or *outbound* NAT [41].

In the first case, since the NAT of client A allows outgoing connections, the latter can connect directly to client B. However, if client B wants to connect to client A, this connection cannot be established directly because client A's NAT will reject any connection request from a host not directly connected to A. The technique known as

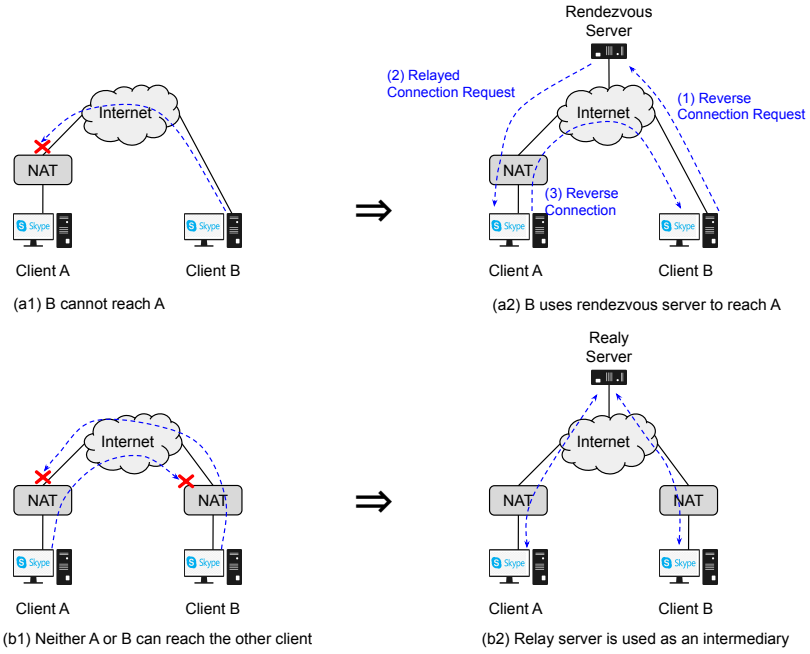


Figure 2.10: Middlebox traversal by Connection reversal and Relaying techniques.

Connection reversal, a form of *reverse engineering*, is then used and is based on the use of an intermediate server S called *rendezvous server*. The server S is not behind a NAT, is well-known by both clients, and is directly connected to client A. Client B will then send a *reverse connection* request to the server S, which relays it to client A. The latter will therefore connect directly to client B successfully since its NAT allows outgoing connections.

In the second case, i.e. both clients A and B are behind a NAT, the situation gets more complicated. Neither client can connect directly to the other, as their respective NATs block all incoming connections. In this case, the technique employed is that of *relaying*, which consists of using a third client C as a *relay server*. Client C is not behind a NAT and is known to both clients A and B. The most commonly used protocol for relaying is the TURN signaling protocol [42] which is an extension of the STUN protocol [43].

These scenarios are illustrated in Fig. 2.10 that is an adaptation from [41]. The major drawback of signaling is undoubtedly the additional delay it can introduce on the data transfer. Moreover, in the case of relaying, the relay server could become a concentration point for too many relay requests and gave in to a breakdown under unacceptable overload.

Dissimulation. As its name suggests, dissimulation involves "hiding" the protocol data either with the help of encryption or by encapsulation of the protocol data in the packets that take the format of one of the protocols accepted everywhere on the Internet: TCP or UDP (see Fig. 2.11). For instance, QUIC uses both techniques to ease traversal of middleboxes and to reduce their

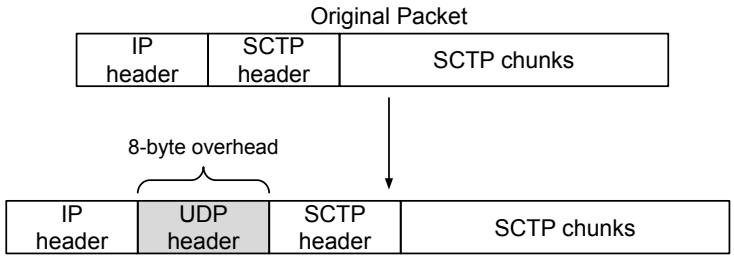


Figure 2.11: Example of encapsulation of SCTP over UDP.

influence on the protocol functioning. Nevertheless, encryption or over-encapsulation will inevitably result in increasing the number of bytes per packet, thus introducing overhead.

Fallback to TCP or UDP as a last resort

There is finally the case where nothing works because the middlebox is not just a "dumb" *outbound* NAT but has built-in complex algorithms allowing it to detect reverse engineering and relaying techniques or "even better", is combined with firewalls that simply remove suspicious packets. In such situations, a final approach is to fall back to a protocol accepted by almost all middleboxes on the Internet (mostly TCP or UDP). If we go back to example 1, when the middleboxes do not allow the Skype default protocol to pass, the protocol implemented by Skype systematically switches to TCP. The purpose in case of systematic fallback is to avoid a complete failure of the application: a minimal quality service is better than a faulty service.

All in all, through extensive measurements, the authors of [13] have 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. Nevertheless, this thesis preconizes and integrates a systematic fallback to TCP or UDP in order to prevent complete failure in case of rejection during middlebox traversal.

Table 2.1 summarizes a set of selected Transport protocols and architectures that we discussed above. We refer readers interested in the chronological history of the Transport layer protocols and architectures to [21, 22].

	MPTCP	**QUIC**	**ETP**	**UTCP**	**NEAT**
Deployment Space	Kernel	User	User	User	User
Protocol-agnostic API	✗	✗	✗	✗	✓
Legacy Appli Support	✓	✗	✗	✗	✗
Middlebox Traversal	✓	✓	✓	N.A	✗

Table 2.1: *Selected* Transport Layer protocols and architectures.

2.3 Thesis Approach and Positioning

In the light of the lessons learned from the limitations of the previous work, this thesis proposes an approach that jointly *(i)* treats the deployment issues within the end-systems, and *(ii)* facilitates the adoption of new protocol solutions by (aware as well as legacy) applications. We achieve our approach within VTL, a Virtual Transport Layer.

VTL envisions a *stable*[5] *framework* that must allow a ***dynamic deployment/loading of protocols*** as would do a typical Web browser that enables plugins insertion to extend its functionalities and features. Therefore and taking into account the points discussed in Section 2.1, VTL should provide a safe and isolated runtime environment for protocol mechanisms in the fashion 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. In straightforward terms, VTL should conciliate the *performance* and *flexibility* on one side, and on the other side, guarantee the *safety* and *isolation* of the end-system OS. Our technical choices have focused on the in-kernel eBPF Virtual Machine (VM) [3] as the runtime environment for protocol mechanisms to fulfill these conceptual requirements. Thanks to its integrated verifier (Fig. 3.3 and Fig. 3.6), 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 the kernel's behavior: protocol components could be inserted at runtime without the need to recompile and upgrade the OS. Last but not least, VTL follows three main ***design principles*** described below.

5: According to David D. Clark, the stability of a platform defines its ability to make innovations possible [8]. A stable platform provides the necessary tools to allow other stakeholders (users, applications, etc.) to introduce new functionality. The most legendary example of a stable platform is the IP layer, which by design has enabled the deployment of applications regardless of their type. Another example is the "Google store" or "App store" which allow developers to innovate by building and deploying their different applications.

Separation of protocol from aware-application. As already stated, in the standard socket API, the legacy application specifies the protocol to be used at the design-time. This leads to a binding and a 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 of the application's 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 requirements (in terms of Transport features/services associated with QoS parameters) instead of the specification of the protocol to use. The choice of the most appropriate protocol mechanisms to satisfy the application's needs is left to VTL. This principle is in the same line with the TAPS standard [6] but goes far by providing seamless

support to legacy applications and a stable framework for runtime deployment of protocol mechanisms.

Transparent integration of legacy applications. To recall, the major existing applications use the standard socket API to consume Transport layer services. Most of the time, application programmers are often unwilling to switch from the standard API socket to the API of any new protocol or architecture. Therefore, to ease and stimulate its adoption, VTL provides transparent support to legacy applications i.e. those applications should consume Transport layer services without the need to rewrite them. This may be performed by seamlessly redirecting the socket API invocations of legacy applications towards the protocol-agnostic interface.

Protocol modularization. In line with the conceptual choices adopted in the ETP protocol solution, we split any protocol in a set of small units called Transport Function (TF) that are packaged in deployable/pluggable software wrappers (such as eBPF programs, loadable kernel modules, or Docker containers). The distribution of TFs interacting on both sides of the communication leads to providing part of the final end-to-end service. This principle eases per-session protocol configuration and reconfiguration. Further, the integration of 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 align with the evolution of network conditions or the application's requirements by replacing or inclusion of new TF that provides more appropriate features in the novel context; and *(iii)* the *customization/efficiency*, i.e., the capacity to tailor the protocol to the application's requirements and therefore reduce overheads by the elimination of unnecessary services.

2.4 Conclusion

This chapter provided a review of the literature on the Transport layer and its ossification issues. We enumerated the different elements that hamper the introduction of innovation within the Internet's Transport layer during this review. We show that these elements formed the so-called vicious circle phenomena that could be summarized as follow: *(i) application programmers* are unwilling to use a new Transport protocol that is unlikely to work end-to-end; *(ii) operating system (OS) developers* will not implement a new Transport protocol if application programmers do not express a need for it; and *(iii) middleboxes vendors* (somewhat network operators), will not add support if the Transport protocol

is not in mainstream operating systems; the new protocol will not work end-to-end because of lack of support in middleboxes. In Section 2.2, we discussed the most relevant previous works that address the Transport layer's ossification issues and pointed out the shortcomings of those works. Finally, learning from this preliminary review of the literature, we closed the chapter by introducing the requirements and design principles of a new Transport layer system that we called VTL for **V**irtual **T**ransport **L**ayer. VTL aims to enable the deployment of any new Transport protocol, not only in the user-space but also in the kernel-space; it also facilitates the utilization of the deployed protocol by *legacy* and *aware*-applications. In the next chapter, we introduce the VTL system by emphasizing on its dynamic deployment aspects.

Virtual Transport Layer Introduction

3

The previous chapter outlined the limitations of current Transport layer protocols and architectures to address the ossification issues. Moreover, it motivated the need to revisit the Transport layer architecture and provided the guidelines for designing a new architecture of the Internet's Transport layer. Following these guidelines, we designed and implemented a novel Transport layer able to break the vicious circle's substantial consequence, i.e., the Transport layer's sclerosis. Fundamentally, the novel Transport layer *1/* provides a stable framework to accommodate (new) Transport protocols *dynamically* deployed and *2/* exposes a *protocol-agnostic API* to aware-applications in order to facilitate and promote the use of the deployed L4 protocols. Further, it provides the necessary architecture elements and mechanisms that allow legacy applications to leverage the available Transport services without the need for those applications to be rewritten. We will discuss this latter aspect of VTL in Chapter 4.

This chapter, organized into four sections, presents the design and implementation of such a novel Transport layer, which we called VTL (for **V**irtual **T**ransport **L**ayer). It is worth noting that VTL is ***not*** a new Transport *protocol* but rather a new Transport *layer architecture*, which in essence, aims to allow the deployment and the use of *any* (new or existing) Transport protocols such as SCTP, QUIC, and so on. In the first section of this chapter, we introduce VTL by presenting its core concepts. Then, Section 3.2 presents a detailed functional architecture of VTL. This description emphasizes on the aspects of dynamic deployment/integration of L4 protocols within the end-systems' operating system (OS) as well as on the interaction between VTL and aware-applications. The way legacy applications integrate with VTL is described later in the next chapter. In Section 3.3, we evaluate VTL through extensive experiments. Apart from showing the correctness[1] of VTL, this evaluation presents the deployment delay of the protocols and the performances (under VTL) of the deployed Transport protocol mechanisms by taking as reference TCP performances in the same network context. The selected performance metrics are data transfer throughput and latency. Finally, Section 3.4 and Section 3.5 conclude the chapter with a discussion on closely related research efforts to the contribution presented in this chapter and a summary of learned lessons.

1: We define correctness as the ability of the system to successfully perform its primary tasks. It is also known as functional evaluation as opposed to non-functional evaluation that assesses the systems' performances.

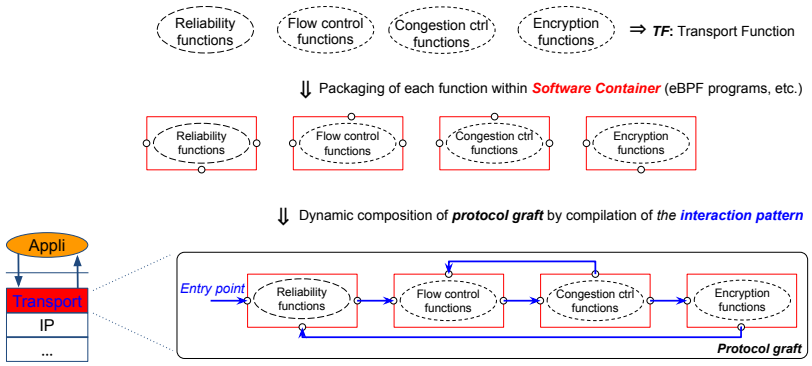


Figure 3.1: Conceptual illustrations of Transport function (TF) and protocol graft.

3.1 VTL Core Concepts

As stated in Chapter 2, we designed VTL around a set of principles, namely the notion of a *protocol composition from a set of basic functions*. This section introduces the fundamental concepts of this principle and presents an overview of the services and features provided by VTL. It is worth noting that the concepts related to a protocol composition are not new and specific to VTL. Similar notions have been introduced in previous work such as CTP [44] or ETP [25] described in Chapter 2. The main goal of the introduction of the below concepts is to show the readers how VTL organizes its data plane for more efficiency in the management of protocol deployment and data delivery.

3.1.1 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. 3.1): reliability functions, flow control functions, congestion control functions, security/encryption functions. 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. 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. 3.3).

Each TF must be made *loadable* or *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 at least two essential 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.

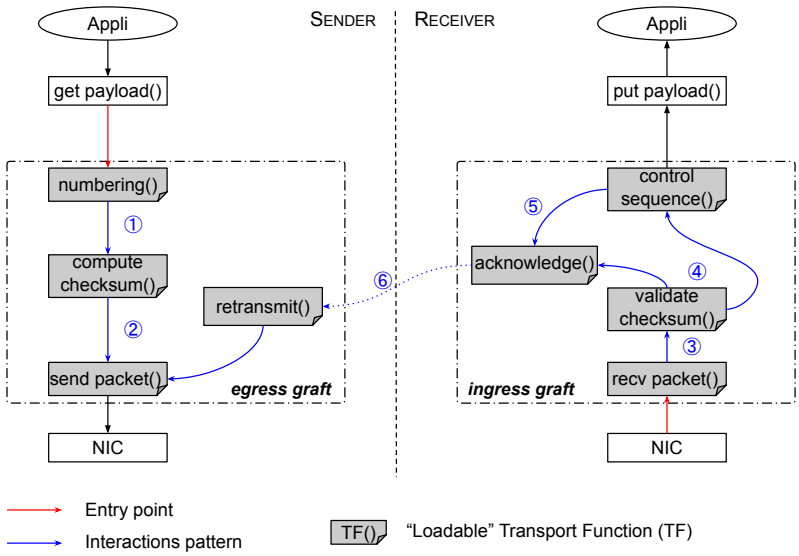


Figure 3.2: Illustration of protocol graft providing an ARQ-based reliable service. Basically, after numbering and checksum adding to the packet's header (①), the packet is sent to the receiver through the NIC (②). The receiver will then control the checksum and sequence number values (③ and ④). The acknowledgment is then transmitted to the sender to signal whether the packet is correctly received (⑤ and ⑥).

3.1.2 Protocol Graft

As illustrated in Fig. 3.1, a *protocol graft* is a list of loadable TFs with their *interaction pattern*. The interaction pattern 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, multipath service, encryption service, and so on. Contrary to a primary 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 ones and provides a comprehensive single or more *Transport service(s)/feature(s)*. The sender side maintains the *egress graft*, whereas the receiver side maintains an *ingress graft*. Fig. 3.2 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. 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.

ARQ is a protocol mechanism that relies on acknowledgments and timeouts to provide reliable Transport service during data transfer. The implementations of its variants are further described in Section 3.3.

3.1.3 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. To do that, VTL provides applications with an access point to Transport services in order to allow the latter *(1)* to express their requirements, *(2)* to send and receive data, and *(3)* to reset their requirements on-the-fly. The

Table 3.1: Considered Transport Services/Features.

Transport features	***Definition***	***Granularity / Scope***
Full Reliability	Data packets are delivered to the receiver without corruption, no duplication, and no loss.	*Single packet* Processing
Partial Reliability	Data packets are delivered with an acceptable but bounded loss rate.	
Full Order	Data packets are submitted to the receiver in the same sequence the sender transmitted them.	
Partial Order	Some out-of-sequence data packets might be delivered to the receiver.	
Encryption	Data packet contents are ***not*** transmitted in plain "text".	
Flow Control	Control the sending rate of sender and receiver.	*Packets Stream* Control
Congestion Control	Control the rate of data sending over the network to prevent the latter collapses under congestion.	
Multipath	Use multiple network interfaces to transfer data.	

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 can consume VTL services without requiring any modification of its code. From an internal point of view, VTL:

- ensures the Transport session establishment following a classical client (connect) / server (accept) model;
- ensures the selection, the negotiation, and the deployment of the appropriate protocol graft;
- and 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[2] 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. Again, as stated above, VTL makes it possible to consider *(i)* the use of protocols that already exist in the operating system and *(ii)* the deployment of protocols (which do not yet exist in the operating system) in the form of user libraries. We will demonstrate these latter points in Chapters 4 and 5 with QUIC and an SCTP userland version.

2: Message-stream-oriented and byte-stream-oriented notions are defined in Section 2.2 of Chapter 2.

VTL allows applications to express their requirements by firstly characterizing the desired Transport services and then associating those Transport services with a set of required or desirable QoS parameters. For instance, a requested Transport feature or service might be expressed in terms of *reliability* (full or partial), *order* (full or partial), *flow control*, *congestion control*, *security*, or *multipath*.

Table 3.2: Considered quality of service (QoS) parameters.

QoS	*Definition*	*Value Type*
Delay	The end-to-end elapsed time between the sending applicative entity has invoked the sending primitive (e.g. `send()`), and the moment the data has been made available to the receiving applicative entity.	scalar in ms
Throughput	The amount of sent or received data per unit of time.	scalar in Mb/s
Loss rate	The ratio of packets that should fail to arrive at the receiver without impact application well functioning.	scalar in percentage

In addition 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_throughput`, expressed in Mb/s), and allowable loss rate (`max_loss`, expressed in percentage). Some requirements defined by applications may be redundant. For instance, an application that requests to set its allowable loss rate (`max_loss`) to zero and simultaneously invokes the use of a fully reliable Transport service. This thesis does not consider these redundant (and somewhat inconsistent) cases. The set of considered Transport services and QoS parameters are provided in Table 3.1 and Table 3.2, respectively.

3.2 VTL: Design and Implementation

In this section, we begin with a description of the main functional components of VTL architecture. Then, we present the background on relevant technical aspects of VTL. For recall, we discussed in Section 2.3 (of Chapter 2) the design requirements that lead to the major technical choices we made. Finally, the rest of the section describes in more details the implementation of VTL, namely the Transport session initiation, the protocol negotiation and deployment, and the data transfer stage between and within the end-systems.

3.2.1 VTL Architecture Overview

We present the main components and workflows of VTL in Fig. 3.3. VTL components are separated between two planes: a *control plane*[3] constituted by a userland library and a *data plane* constituted

3: There is no standard definition of control or data planes. In the SDN context, for instance, the data plane is composed of switches ensuring data forwarding. In contrast, the control plane is responsible for dispatching and dictating the forwarding rules to the switches. In our work, the control plane is responsible for the management and deployment of the protocol functions, and the data plane is in charge of ensuring the execution of the deployed protocol functions by providing the appropriate runtime environment.

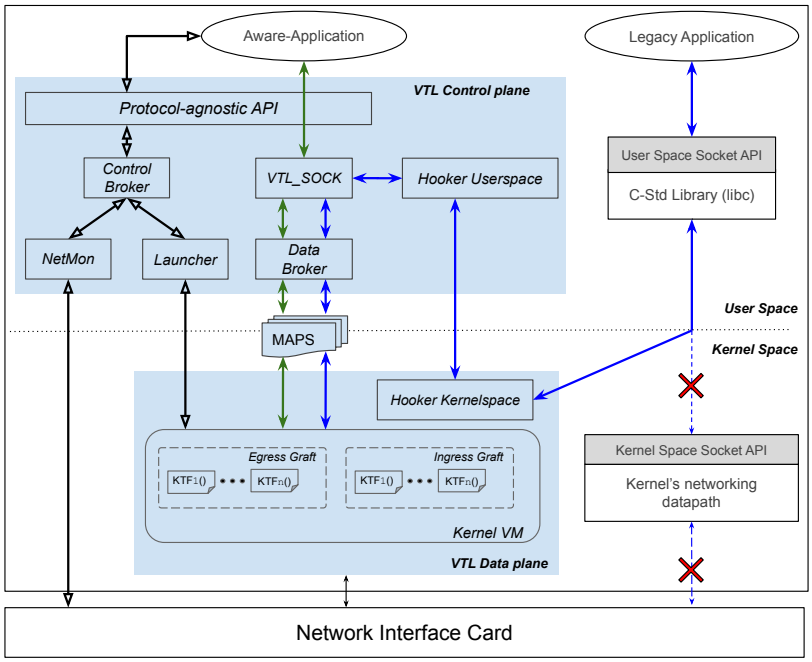


Figure 3.3: VTL system architecture overview. Blue arrows represent the legacy application datapath, green arrows the aware-application datapath, and black arrows the KTFs deployment path.

mainly by the runtime environment of Transport functions (TFs), i.e. the eBPF kernel VM.

Control Broker is the central piece of VTL control plane and its operations during the end-systems' features discovery, session initiation, session maintenance, and session teardown. Among others, it is responsible for ensuring the protocol graft negotiation phase at the end of which the appropriate Transport functions 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*. Prior to each deployment, *Launcher* leverages the *extended* eBPF verifier to ensure that the protocol functions' execution inside the runtime environment is safe for the overall system. 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. Control plane components work together to ensure a (structural) reconfiguration of a session if necessary. Currently, a typical structural reconfiguration loop is triggered by the change in the network state thanks to the quality parameters captured by the *NetMonitor* component which reports feedback to the *Control Broker*. After analyzing the reported network parameters, the *Control Broker* should determine whether a (structural) reconfiguration is necessary. If a reconfiguration is required, *Control Broker* will ask the *Launcher* to remove the old protocol mechanisms and replace them with new ones.

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 [45] for data transmission and/or an XSK socket [46] for data receipt. Briefly, TCP/UDP sockets give access to the Transport layer, a RAW socket gives direct access to the IP layer, and an XSK socket gives direct access to the network interface card (NIC). 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 Transport protocol mechanisms running inside the *eBPF Kernel VM* component. The latter, i.e. the *eBPF Kernel VM*, is responsible for providing a safe and isolated runtime environment for TFs.

Two main actors interact with the VTL system: the *aware-application* and the *legacy application*. In general, an *application* is defined as a computer program running in the user-space of the OS and that uses explicitly or implicitly VTL to send and to receive data in order to perform its business process (e.g., web browser, media player, file transfer, etc.). To recall and more specifically, *(1) aware-application* is defined as an application that should be aware of VTL and uses the provided *Protocol-agnostic API* to send and to receive data, and *(2) legacy application* is defined as an application that uses the standard socket API to consume Transport layer services; the latter should be supported seamlessly, i.e. without any modification of its code.

VTL control plane and data plane share information (ACK/NACK, KTF negotiation state, data packets, ...) thanks to a set of shared *MAPS* (described more in-depth later). These *MAPS* are attached to the root filesystem /sys and serve as a buffer for deployed protocol grafts. VTL defines two main workflows: *KTF deployment workflow* and *data delivery workflow* that we further described in subsection 3.2.5 and subsection 3.2.6, respectively.

3.2.2 Background

This subsection provides the reader with a background on eBPF [3], the core technology on top of which VTL *implementation* currently relies on. In 1992, S. McCanne and V. Jacobson presented the Berkeley Packet Filter (BPF) virtual machine [47] 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* in popular tools such as tcpdump [48] or Wireshark [5]. Recently, Linux introduced an extended version of BPF and named it eBPF, for extended BPF. The name of eBPF no longer reflects the reality and might be confusing because many new features are added to it, and its usage scenarios go far beyond the

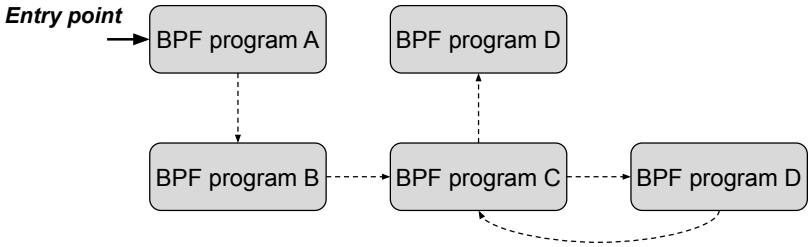


Figure 3.4: eBPF programs chain by tail calls [49].

naive filtering. Indeed, eBPF is used today by many industrials like Facebook or Netflix to perform *tracing*, *monitoring*, *networking*, or enhancing their systems' security. Like its predecessor, eBPF allows injecting bytecode within the OS kernel at runtime, i.e. without having to recompile the OS. eBPF infrastructure is constructed around three major elements: *maps*, *tail calls*, and *helper functions* (see Fig. 3.6).

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. There are several types of maps; each type serves a different purpose. For example, a map of type `SOCKMAP` (more discussed in Chapter 4) must be used to store only sockets' file descriptors. Maps are often attached/pinned to the root filesystem (i.e., `/sys`) to ensure data persistence between successive invocations of eBPF programs.

Tail calls. 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 implementation of modularization. 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. Tail calls concept is illustrated in Fig. 3.4.

Helper functions. Recall the security model used by the kernel to protect itself from user-space programs is based on *system calls* that can only be used by the user-space processes; the processes already running in the kernel, such as eBPF programs, cannot use the system calls. Therefore, the question is: *how can the kernel protect itself from the program running in the kernel-space without limiting the program capacity/functionality*? To face this concern, eBPF introduced the so-called helper functions. 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 functions by eBPF programs is strictly controlled to prevent OS damage. In other words, helper functions allow eBPF programs to interact directly with the kernel in a safe manner. For instance, to perform tail calls, `bpf_tail_call()` helper function may be used

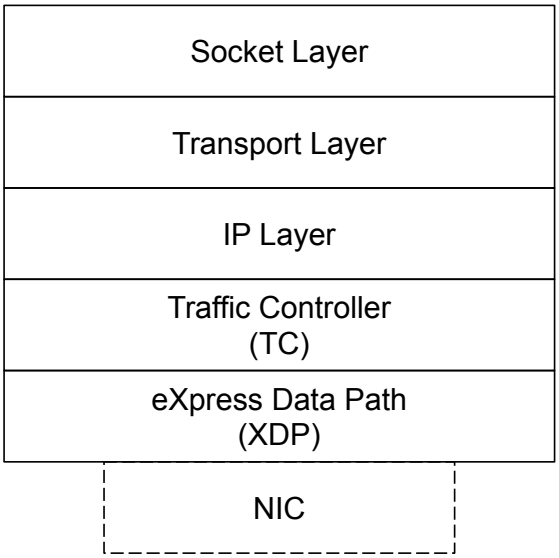


Figure 3.5: Traffic Controller (TC) and eXpress Data Path (XDP) positions in the network stack.

by the ongoing eBPF program to ask the kernel to jump to another eBPF program.

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 operations, etc.). Then, each time the event occurs, the eBPF program attached to it is executed. For instance, eXpress Data Path (`XDP`) [50] and Traffic Controller (`TC`) [51] are two main networking hook points around which we built the VTL system. `XDP` and `TC` reside at different levels of the network stack as depicted in Fig. 3.5. The *type* of eBPF program indicates three primary information: *(1)* the set of helper functions it has access to, *(2)* the data structures (i.e., the maps) that the program is allowed to use, and *(3)* the hook point to which the program is attached. For example, a `SOCKET_FILTER`[4] program type can access and manipulate `sk_buff`[5] structure [52] whereas an `XDP` program type *cannot* do that but must instead use a specialized `xdp_buff` structure [49].

eBPF technology introduces programmability in the Linux kernel by allowing runtime code injection within the kernel. As illustrated in Fig. 3.6, it provides the ability to dynamically add functionalities to the kernel from user-space programs compiled by LLVM/Clang, a user-space compiler. Thanks to the Just-In-Time (JIT) kernel compiler and the eBPF verifier, each functionality is safely added and is efficiently executed. Contrary to kernel patches [23] that permanently modify the operating system (OS), eBPF has the additional advantage of allowing temporary modification of OS kernel behavior. This enables fast prototyping and testing of new features so far as in the case of failures or bugs, the last added features may be "easily" removed to restore the initial and stable state of the kernel.

4: The full name is `BPF_PROG_TYPE_SOCKET_FILTER`. We omit the suffix `BPF_PROG_TYPE` each time we refer to the type of eBPF program.

5: `sk_buff` is the common data structure used by the Linux OS to represent any packet in the kernel.

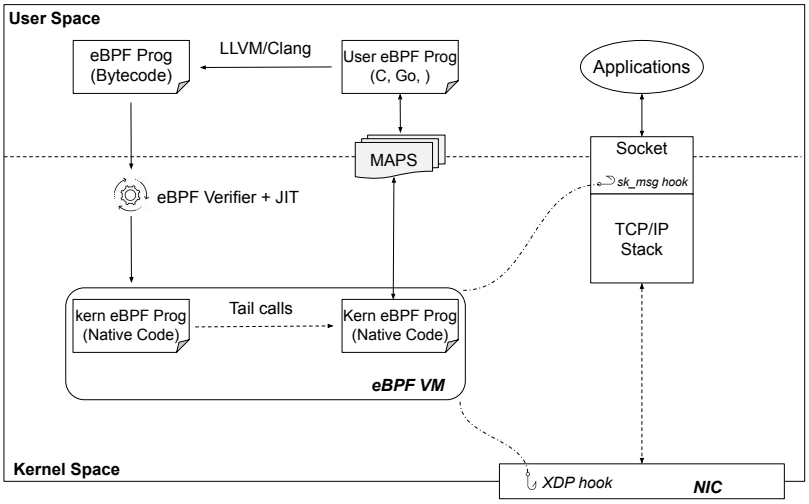


Figure 3.6: eBPF program deployment overview.

To deploy Transport functions in the OS kernel, we first considered the use of Linux Loadable Kernel Module (LKM) approach [53]. As eBPF, LKM allows *hot* plugging of modules in the OS kernel thanks to tools such as `modprobe` [54]. As its name suggested, a kernel module is a regular program that code resides and executes in the kernel-space rather than in user-space. Unfortunately, during our first prototyping, we face the most common issue of kernel module utilization: the whole system frequently crashes at the slightest mistake; in other words, 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, kernel modules 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.

3.2.3 Digging into VTL Implementation

In this subsection and the following, we begin by diving more in-depth on the technical aspects of VTL design. VTL leverages and combines two kernel subsystems: eXpress Data Path (`XDP`) and Traffic Controller (`TC`). The aim of their association is *(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 legacy network stack. Therefore, on its egress path, VTL places `TC` hooks under the IP layer in order to process all outgoing packets as soon as they left the kernel network stack. To reduce the legacy network stack's overheads on the transmission path, 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`

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()` helper

Table 3.3: Main added helper functions within the VTL system.

hooks to the network interface card (NIC) 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 altogether remove the legacy network stack's overheads at the receipt. `TC` and `XDP` are incorporated into eBPF infrastructure, which allows using its verifier to guarantee the deployed bytecode are safe and then reduce most security concerns and end-system crash risks.

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 as soon as they arrived at the network interface card);
- ***(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 connection 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 3.3 summarizes the set of helper functions added by VTL.

3.2.4 Aware-application Session Initiation

Obtaining VTL Socket. In order to transmit and receive data, the aware-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 (see Fig. 3.8) and triggers the deployment of the ***canonical graft*** (described in Section 3.3). 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 uses to send and receive its data. The canonical graft's purpose is to allow the application that does not require specific requirements to instantly send and receive its data without additional overheads and unnecessary delays of the negotiation stage. For applications with specific requirements, the canonical graft is used to conduct the KTFs and protocol grafts negotiation stage. Additionally, the *ingress* canonical graft is useful to conduct a stateful runtime reconfiguration of protocols (see Section 3.3).

Protocol Grafts Negotiation and Deployment. Protocol grafts negotiation process between a sender and receiver is shown in Fig. 3.7. In case of successful negotiation, it ends up with the deployment of the suitable KTFs to satisfy the application's requirements. Each side of the connection maintains its own map named `qos_nego_MAP`. Each index or key of the `qos_nego_MAP` associates a value containing the file descriptor 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 side: the client of the negotiation***. Aware-application that requires specific Transport services and QoS defines them 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 pre-forming, *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 leaves the IP layer, it is intercepted by `egress_tf_sec` which, based on the *gid* field, may determine if the intercepted packet is a negotiation one or not. When the *gid* value is not set (the 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, i.e. the *gid* value is not `NULL`, the packet is a negotiation one and `egress_tf_sec` sets the packet *type*

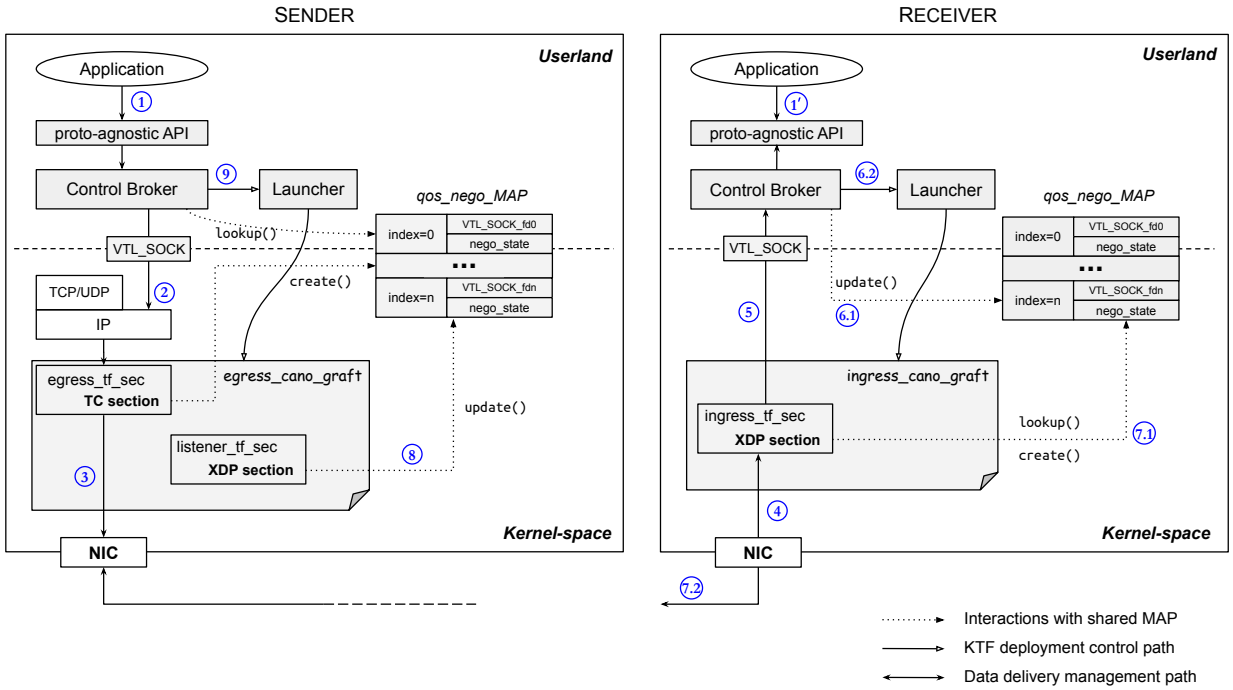


Figure 3.7: Protocol grafts negotiation process under VTL system.

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 the 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` in the map), *Control Broker* triggers the deployment of the 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, 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 receipt 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 requested ingress graft deployment 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 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 it does not.

3.2.5 KTFs Deployment Workflow

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. *Control Broker* component handles the KTF deployment that can be triggered by three events:

- *(i)* an invocation of the regular socket API or the protocol-agnostic API by applications,
- *(ii)* a request for a connection from remote end-system received on VTL socket (*VTL_SOCK*), and
- *(iii)* a change in network condition reported by the *NetMonitor* component.

When *Control Broker* requests the deployment of a specific KTF stored as a user-space object file, *Launcher* component picks it in the *KTFs pool* and starts its loading within the eBPF VM. *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 userspace compiler (Clang/LLVM) during the deployment of protocol functions (see Section 3.3). Before its effective loading, a KTF is checked by the verifier that performs a series of *verifications*[6] 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 in-kernel compiler (JIT) 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.

6: 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; and *(iv)* the number of instructions in the KTF that must be under 1M (4096 for all Linux version prior to v5.2.0).

3.2.6 Data Delivery Path

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 data transfer may be preceded by a protocol graft negotiation stage that ensures that the appropriate KTFs (i.e., satisfying application needs) are deployed and ready to process incoming and outgoing traffics. *Data Broker* controls data moving by configuring and providing to the application a VTL socket

that, as previously stated, emulates either a RAW socket for data emission and/or an XSK socket for data receipt (see Fig. 3.8). After having obtained a VTL socket, a VTL aware-application is able to use the protocol-agnostic interface to send and to receive data.

Fig. 3.8 depicts how internally the VTL socket transfers data through its associated buffers; for the sake of clarity, *Data Broker* and *protocol-agnostic API* components are not shown in this figure. During data moving, the interactions between the application and VTL system are performed asynchronously thanks to a pair of buffers at the transmission (*Tx buff* and *skb buff* in Fig. 3.8) as well as at the reception (*Rx buff* and *umem* in Fig. 3.8). *Rx buff* and *Tx buff* are useful to ensure the protocol graft 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.

There is no intermediate IP layer processing at the packets' reception, and the received data should 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 [46] features to associate two ring buffers to the *umem* buffer: the *fill ring* and the *Rx ring*. The former is used by *Data Broker* to pass the ownership of the packet buffer to the kernel (ⓘ in Fig. 3.8) whereas the kernel uses the *Rx ring* to pass the ownership of packet buffer to *Data Broker* (ⓘⓘ in Fig. 3.8). 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

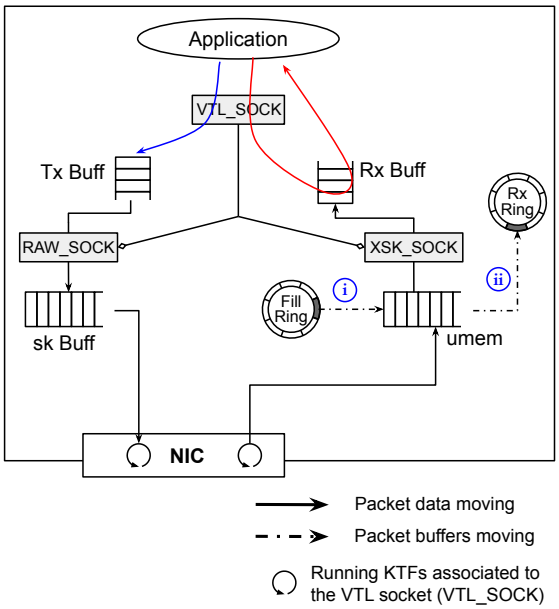


Figure 3.8: Data moving between application and VTL socket and its associated buffers.

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*.

3.2.7 VTL Aware-application Session Summary

This section summarizes a typical Transport session of an aware-application by a description of the *Protocol-agnostic API* and its interaction with applications. *Protocol-agnostic API* component ensures the principle of the separation between application and protocol. It is a shared library used by aware-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 describe the currently implemented functions through a typical function call flow as illustrated in Fig. 3.9. Furthermore, Table 3.4 summarizes the way applications should specify parameters when using the *protocol-agnostic API*.

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. At this stage, thanks to canonical grafts *associated* with the newly created VTL socket, the application (without any special requirements) 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 having specific requirements may call into `vtl_negotiate()` to transmit its needs to the VTL system. At the receiver side, the application must invoke the blocking function `vtl_validate()` to indicate to the VTL system that it is waiting for a graft negotiation stage result. Both functions `vtl_negotiate() / vtl_validate()`

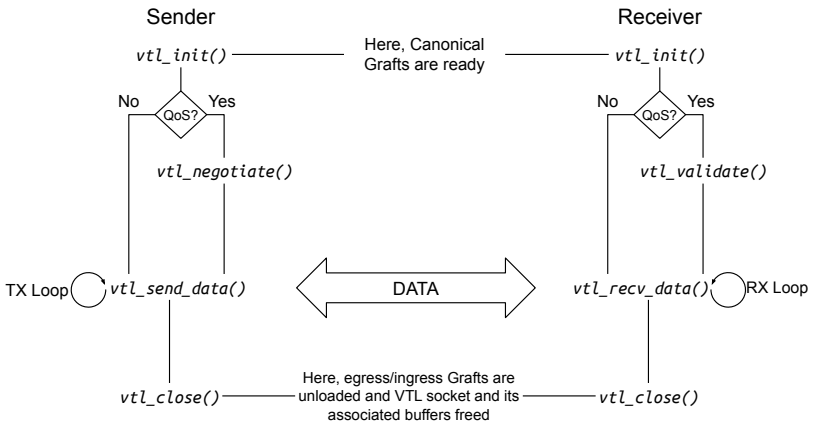


Figure 3.9: Typical function call flow by VTL aware-applications for data Tx/Rx.

Table 3.4: 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 outcome for the application.
`vtl_send_data()`	`vtl_sock, buffer, buffer_size`	Send application's payload data and retrieve the size of written data.
`vtl_recv_data()`	`vtl_sock, buffer, buffer_size`	Fetch application's payload data and return the read data size.
`vtl_close()`	`vtl_sock`	Close VTL socket and free its associated resources (buffers, KTFs, etc.)

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 VTL socket as well as *(ii) unload* the KTFs of grafts associated with the socket.

3.3 Carried Out Use Cases and Performance Evaluation

This section aims to demonstrate the ability of the VTL to allow the dynamic plugin of KTFs in response to service requests. We also assess VTL outlooks' benefits, thanks to the evaluation of the performances resulting from the deployment (under the VTL) of Transport protocol mechanisms. The reference we used to discuss the obtained results is TCP (cubic) that we evaluated in similar network conditions.

In terms of developments, we revisited and implemented, *from scratch*, a set of protocol mechanisms well-known in the literature. As stated above, the goal is to demonstrate the dynamic deployment ability of VTL and show how the L4 protocol functions can be implemented as eBPF programs. All implemented mechanisms have the same structure. At the sender side, the egress graft maintains two KTFs: one `TC` section named `egress_tf_sec` that ensures the processing of all outgoing VTL packets, and one `XDP` section named `listener_tf_sec` whose job is to process acknowledgment packets. At the receiver side, the ingress graft runs a single KTF

that consists of an XDP section named ingress_tf_sec and that is sufficient to process incoming packets as well as to send acknowledgment if necessary thanks to the verdict XDP_VTL_ACK. Code Listing 1 illustrates a *template* of an egress graft part of a protocol. Subsection 3.3.1 introduces some L4 protocol mechanisms we implemented. Later, in subsection 3.3.2, we present a runtime (structural) reconfiguration technical approach under VTL. Finally, subsection 3.3.3 and subsection 3.3.4 present the configuration of the experiments environment and discuss the outcomes of the evaluations.

3.3.1 Implemented KTFs and Grafts

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 do not have special requirements 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. Canonical grafts are deployed at the creation of a new VTL socket to which the KTFs composing the canonical grafts are associated by default (go back to Fig. 3.8). 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 for the outcome of negotiation handled by *Control Broker*, transmits an acknowledgment to the sender, and pursues the next packet's processing. Moreover, the *ingress* canonical graft has the property to be systematically activated (resp. deactivated) when there is no other XDP section running on the network interface driver (resp. when a new XDP section is attached to the network interface driver). This latter property is essential to ensure a stateful reconfiguration of KTFs (discussed in subsection 3.3.2).

ARQ Reliable Graft Based on Go-back-N. Go-back-N is an optimized version of stop-and-wait algorithm [55]. Instead of sending only one packet at a time, the 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

Listing 3.1: Template of protocol graft and Kernel Transport Function (KTF).

```
#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 ***/
}
```

sequence numbers as opposed to stop-and-wait algorithm where there is no need for numbering packets[7] . Therefore, at the receiver, in addition to data integrity validation, `ingress_tf_sec` makes sure that the packet is in sequence before positively acknowledging it.

7: Note that the assumption is made here that there is no loss on ACK packets.

Selective Repeat (SR) Protocol Graft. Selective Repeat protocol is a bit more complex and optimized version of Go-Back-N protocol mechanism. Like the latter, Selective Repeat mechanism allows the sender to transmit at a time $N > 1$ packets without waiting for acknowledgment from the receiver. N is the sender window size. But, contrary to Go-Back-N, a lost packet is retransmitted alone rather than retransmit all packets of the window from that point. At this end, the receiver side (`ingress_tf_sec`) accepts and buffers out-of-order but not corrupted packets, waits for retransmission by the sender of lost/corrupted packets of the window before delivering the out-of-order packets to the application.

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 (such as multimedia applications). This considerably reduces the transmission delay with less impact on the proper execution of the application.

Fig. 3.10 illustrates the partial reliability concept under video streaming where each packet data carries one image. The assumption is made that the loss of one data packet (i.e. image) every ten images is acceptable because it is not perceptible by the human's eye. In the first case, only one image (I3) is lost, the data packet containing that image is not retransmitted. In the second scenario, the images I3 and I8 are lost. The first lost data packet is not retransmitted but to fulfill the *MAX_LOSS* requirement (*10%* here), the second lost data packet (I8) is retransmitted.

3.3.2 Runtime (Re)configuration of Grafts Use Case

VTL leverages eBPF infrastructure dynamic reloading features to guide runtime configuration or reconfiguration of protocol grafts. It consists of a dynamic deployment / replacement of KTFs attached to `TC` or `XDP` hooks without application outage. Reconfiguration may be performed either to end up a successful protocol grafts negotiation process or when *NetMonitor* component reveals change within the network that requires adaptation actions. 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 (see Fig. 3.11). 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 network interface driver 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 does not 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 (see Fig. 3.8) before old ingress graft unloading, VTL leverages the systematic activation properties of the *ingress* canonical graft (when there is no `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.

3.3.3 Testbed Setup and Methodology

We implemented and evaluated VTL under Ubuntu distribution running Linux 5.3.5. The goals of carried experimentation were to evaluate *(i)* the correctness of VTL, i.e. its protocol deployment and reconfiguration capabilities, *(ii)* the performances of VTL especially in terms of deployment delay, and *(iii)* the performances under

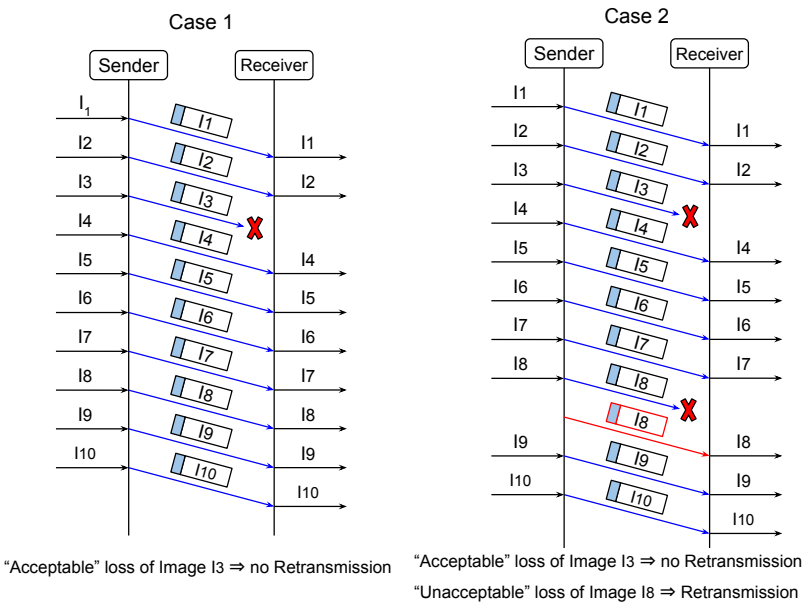


Figure 3.10: Partial Reliability illustration under video streaming with acceptable loss rate set to *MAX_LOSS* = 10%.

VTL of the implemented KTFs and protocol grafts namely in terms of data transfer latency and throughput. 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 [56]. Indeed, `netem` allows us to apply random delay on the network link. The delay is uniformly distributed with mean value at the fixed RTT (100 ms in this experiment). This property is useful to simulate the delay variations that occur in the Internet. Finally, combined with the delay and the bandwidth, the random loss applied to the link permits us to assess the protocol reaction to data packet losses and the way this reaction could affect the final end-to-end performance achieved by the protocol and perceived by the application. Unless otherwise stated, the network parameters used during experimentation are reported in Table 3.5. Each host is equipped with Intel Core i7-7500U CPUs, 3.8 GiB RAM, and Qualcomm Atheros QCA6174 NIC driver.

Further, we built 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 ranging from a simple 4KB file text to more than 32MB video files. In order to capture outliers and thus avoid biases, for each metric evaluated, we repeated the experiment enough (at least 300 trials) and observed the standard deviation of the sample. The mean of the observed sample is taken only for a relatively small standard deviation. As already stated, the reported evaluation of the legacy TCP (Cubic) is provided only as a reference to discuss the evaluated KTFs and protocol grafts.

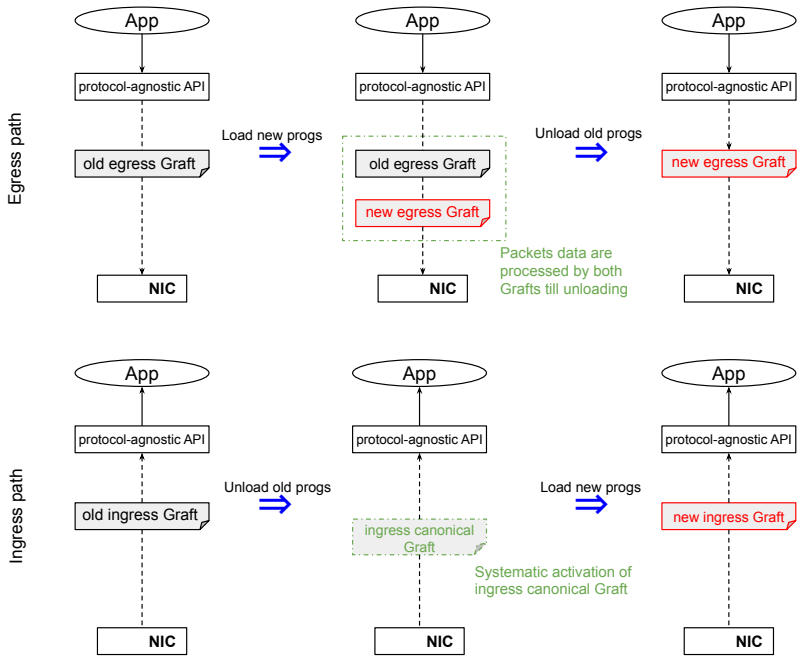


Figure 3.11: Protocol graft stateful reconfiguration actions. To ensure no packet loss, the egress path may generate moderate overheads during stateful reconfiguration.

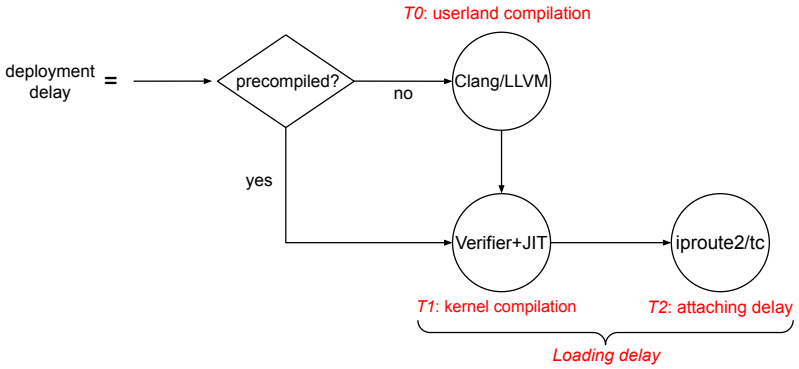


Figure 3.12: KTF deployment delay breakdown. Precompiled KTFs eliminate Clang/LLVM compilation delay (T0). T1 and T2 represent the delay of KTF loading inside the eBPF VM.

Therefore, this is not an absolute comparison with TCP so far as a fair comparison would (among other requirements) require a large scale configuration and deployment. 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 value 253 (that corresponds to the hexadecimal notation `0xfd`) [57].

3.3.4 Microbenchmarks

KTF's Size and Graft Negotiation Delay. Basically, a connection-oriented Transport session takes place in three stages: connection establishment, data transfer, and connection closing (Section 2.2). Under VTL, the delay of session establishment is the amount of time required to negotiate and deploy protocol grafts. The delay of grafts negotiation is an important metric because it indicates the opportunity of whether or not to consider a dynamic deployment. Long negotiation delay could be a real limit to the practicality of the dynamic deployment, especially for latency-sensitive applications. Therefore, the first step of our evaluations consisted in assessing 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. 3.13 illustrate essentially the delay of KTFs deployment within the end-system OS. We compute delays with the help of `time` tool [58].

Min RTT	Link Capacity	Loss Rate
10 ms	16 Mbps	0 - 5%

Table 3.5: Configurations of Network Testbed.

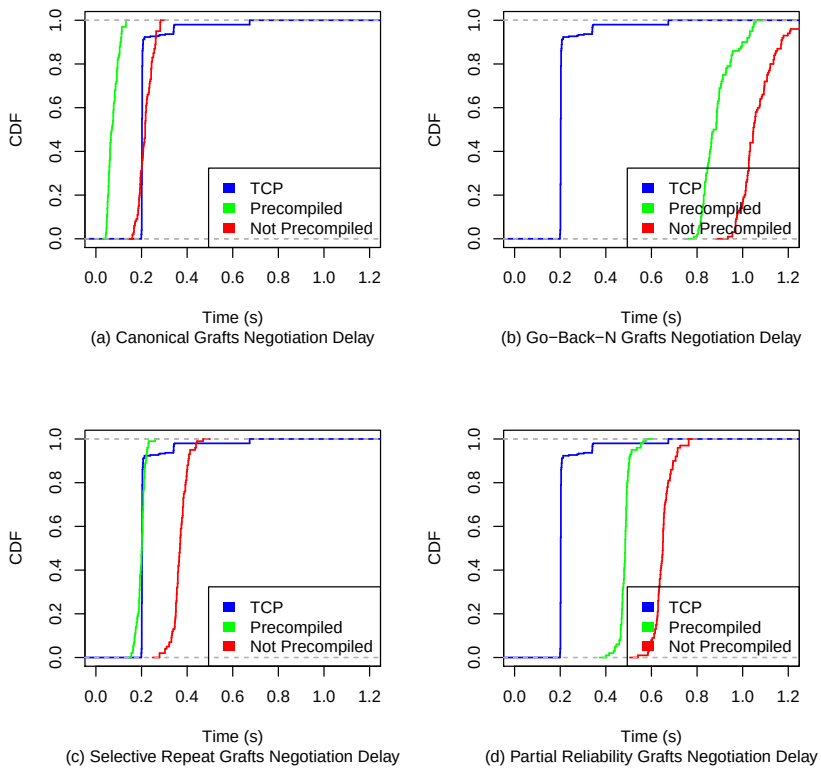


Figure 3.13: KTFs and Graft negotiation delay. Precompiled grafts (green curve) considerably reduce protocol graft negotiation delay. As a reference, the curve in blue illustrates the latency of TCP's 3 way-handshake connection setup with RTT at 100 ms and loss rate set at 2%.

As illustrated in Fig. 3.12, 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 note 6 in Section 3.2) and the `XDP` (resp. `TC`) hooks attaching performed by `iproute2` [59] (resp. `tc` [51]) 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 precompile 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. 3.13 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 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.

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**

Table 3.6: The code complexity of the implemented grafts. It shows the number of source lines of code (SLoC) and the sizes of non-precompiled (source file) as well as precompiled (object file) grafts

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 3.6 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 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.

The SLoC indicates the number of instructions in the source code. It is commonly used to estimate the complexity of the code and the effort required to produce the code. The SLoC can also be used to get information by the program size.

Data Moving Performances: latency & throughput. In addition to the session establishment latency represented by the graft negotiation delay, we evaluated the data transfer performances of each implemented protocol graft under the streaming of files of different sizes. The evaluations were performed within several network conditions by the variation of the network loss rate. The results are reported in Fig. 3.14 and show the file completion time (**FCT**[8]) and the data transfer speed rate. The acceptable loss rate in partial reliability graft is set approximately at 10%. According to [25], this is the maximum loss rate that can be resorbed by adaptive coding [60] for some applications such as multimedia transfer.

8: We define FCT as the amount of time elapsed between the first packet sent and the last packet received during the Transport session.

When there is no packet loss in the network, all protocol grafts have almost equal performances (i.e. FCT and throughput). For small files (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

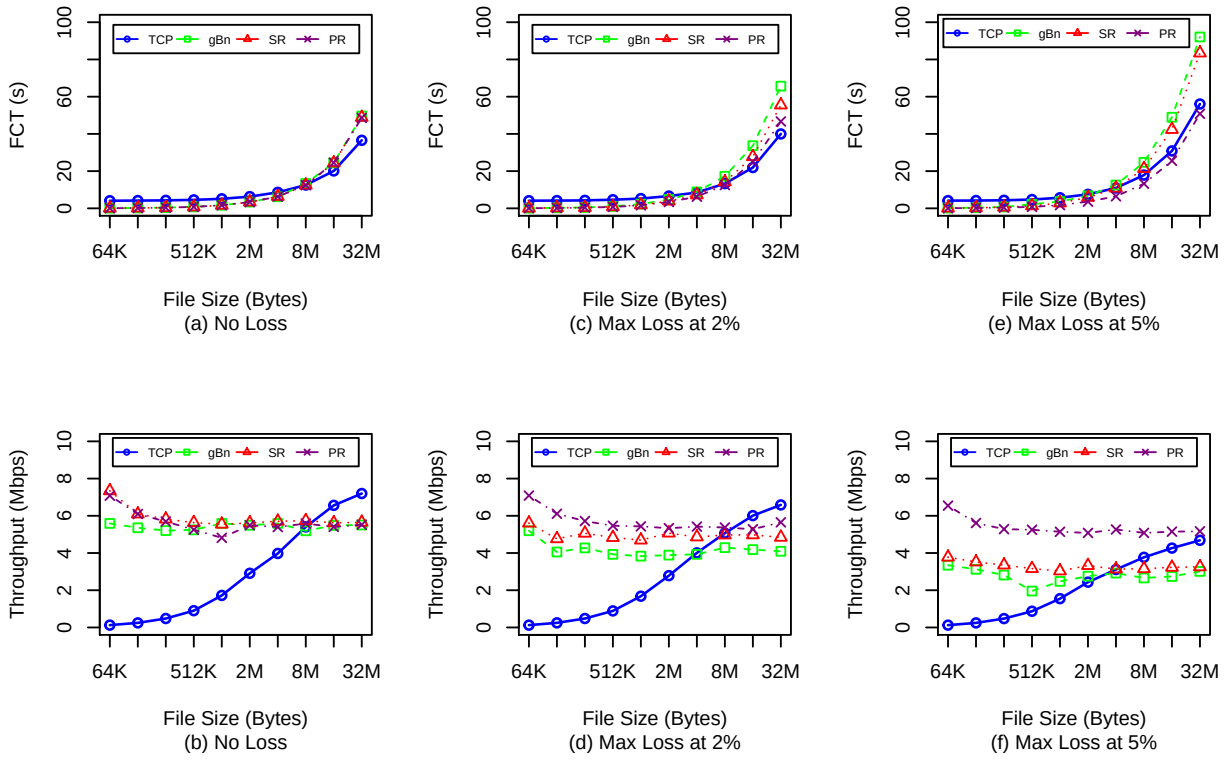


Figure 3.14: Data moving performances under various network conditions of protocol grafts: Go-Back-N (gBn), Selective Repeat (SR) and Partial Reliability (PR).

note in Fig. 3.14 (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[9] (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 $RTT300ms$, it systematically triggers the replacement of Go-Back-N by the partial reliability protocol graft with *MAX_LOSS* keep at approximately 10%.

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

9: The window size is definitely not the throughput, but their variations trends are similar. Indeed, $thgpt = (1 - loss\_rate) * win\_size$.

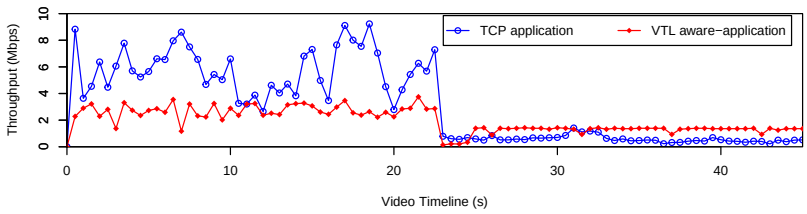


Figure 3.15: Throughput variation of VTL aware-application in changing network conditions.

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. 3.15) show that VTL is able to adapt to the network conditions in order to limit performance degradation. Indeed, we can observe in Fig. 3.15 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.

3.4 Closely Related Work and Discussion

There are some works in the literature that relies 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 STP [61] proposition. 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 [62] that are exchanged remotely between end-systems. STP prototype is written in Cyclone [63], a safe dialect of C language.

More recently, PQUIC [64] introduced a prototype of a framework able to dynamically extend QUIC protocol by loading at runtime new Transport protocol plugins that contain the code implementation of the mechanisms. PQUIC relies on a *user-space version* of eBPF infrastructure. Nevertheless, STP and PQUIC keep the binding to a *specific* protocol, TCP in the case of STP and QUIC in the case of PQUIC. That is to say, they do not provide a protocol-independent API to applications: this prevents applications from the use of new dynamically added functionalities if these applications are not written for the specific protocol and its extensions. As discussed in Chapter 2, this might present a limit to the use of these solutions. In fact, that is already the case with STP that was released several years ago but is not available on the Internet.

3.5 Conclusion

In this chapter, we introduced VTL, a system that can timely and effectively deploy Transport protocol functions within the OS kernel and ensure their flexible usage by applications. We implemented VTL by relying on a combination of `XDP` and `TC` Linux subsystems, part of the eBPF infrastructure. The use of eBPF allows VTL to ensure the extensibility and flexibility of the Transport layer without sacrificing the isolation and the safety of the end-system OS. To evaluate VTL, we implemented *from scratch* a set of protocol mechanisms. During our experimentation, we found that VTL can quickly deploy protocol functions (KTF), especially when the deployed KTFs are precompiled and stored in a dedicated repository. Further, we evaluated the implemented protocol mechanisms' performances in terms of the data transfer latency and data speed rate. Taking reference to the legacy TCP, the results showed that implemented protocol mechanisms present good performances when executed under VTL.

This chapter focused on the dynamic deployment aspect of VTL and the way it interacts with *aware*-applications. In the next chapter, we will present how VTL integrates into a *transparent* way legacy application and allows the latter to use any Transport protocols other than the standard TCP.

Transparent Integration of Legacy Applications

4

Till today, TCP remains the de facto L4 protocol of the Internet's Transport layer despite its well-known limitations in different contexts [65]. To address TCP limitations, a plethora of propositions have emerged in the academic literature and in the industries, from the early IETF standards such as DCCP [66], SCTP [67], etc. to more recent proposals like DCTCP [68], QUIC [26, 27], etc. All of these alternatives to TCP, including its own extensions such as Hybla [69], have seen limited use due *in part* to the weakness of the *socket API* (as discussed in Chapter 2).

To address this limited use of L4 protocols other than TCP, one possibility is to replace the *socket API* with a generic *protocol-independent* or *service-oriented API* so that the application no longer chooses the L4 protocol to use neither during its development nor during its execution. Our first contribution, presented in the previous chapter, is in line with this approach that we enhanced by providing a stable system called VTL that enables dynamic deployment of Transport protocol functions within the end-system OS. The aim of the service-oriented approach is to break the dependency (namely the *static* binding) of the application to a *single* protocol. This approach is promising and is currently promoted by the IETF working group TAPS [6]. However, a major question remains: *how to port existing TCP applications to the new API*? Right now, the answer to this question will require to modify the legacy applications. Based on the lessons learned from the vicious circle (see Chapter 2), we argue that the need to rewrite the legacy applications could be a barrier to the adoption of the new service-oriented API. This should result in *limited use* of *(1)* the protocols integrated within the VTL, as well as *(2)* the VTL system itself. To prevent this eventual limited adoption, we proposed in this chapter an approach that permits to replace at runtime TCP by another protocol *X*. We realized it in a *transparent* way to TCP applications, i.e. there is no need to modify the code of the applications. The approach is implemented by the *Hooker* component of VTL (go back to Fig. 3.3 for VTL architecture overview).

This chapter is organized as follows. We first outline in Section 4.1 the state of the art of existing approaches similar to our proposition. In order to apprehend the introduction of *Hooker*, subject of Section 4.3, we provide a background on TCP and the socket API in Section 4.2. Then, in Section 4.4, we carry

out thorough evaluations of the *Hooker* component in order to assess the functional properties as well as the performances of *Hooker* namely the delay of the TCP replacement and data redirection operations. Finally, Section 4.5 concludes the chapter and motivates the contribution presented in the next chapter.

Note: This chapter's reading cannot be dissociated from that of the previous chapter (at least from subsection 3.2.2). In the following, we assume that the reader has a basic knowledge of eBPF and the main elements of its architecture, notably the maps and hook points.

4.1 Related Work

Long time ago, before being superseded by the TCP/IP standard model, the standard reference model OSI was popular too as the TCP/IP model. Organized in several layers as the TCP/IP model (see Fig. 4.1 (b)), its dominant Transport protocol was TP4 [70] different from the TCP protocol in terms of Transport services specifications. Internet, more precisely ARPANET [71], was then plenty of two major kinds of systems: on one hand OSI end-systems and on the other hand TCP/IP end-systems (see Fig. 4.1 (a)). To ensure the interconnection between these different types of systems, the need for host-level (i.e. Transport-level) interoperability appeared. To fulfill this requirement, in 1986, I. Groenbaek proposed, in [72], *protocols converter* illustrated in Fig. 4.1. The fundamental idea of protocols converters is to split the Transport session in two: TCP session at one side and TP4 session at the other side. This is one of the first steps towards the violation of the end-to-end principle described in Chapter 2. Since many researchers followed and extended Groenbaek's idea leading to what is commonly known today as *proxy* [73, 74].

Like a protocol converter, proxy is a kind of relay equipment that ensures the translation between two different protocols of the same layer of TCP/IP. In addition to the primary task of protocol conversion, a series of proxies known as PEPs (Performance Enhancing Proxies) [73] are able to improve the performance of several protocols such as TCP over satellite links. Albeit most of the proxies operate at the Internet's Transport layer, there exist also several kinds of proxy operating at the application layer namely web proxies such as Squid tool [75] that targets and enhances HTTP(S) protocol [76].

While proxies were originally developed to ensure interoperability between different protocols and to improve these protocols' performance, it soon became apparent that their use could also

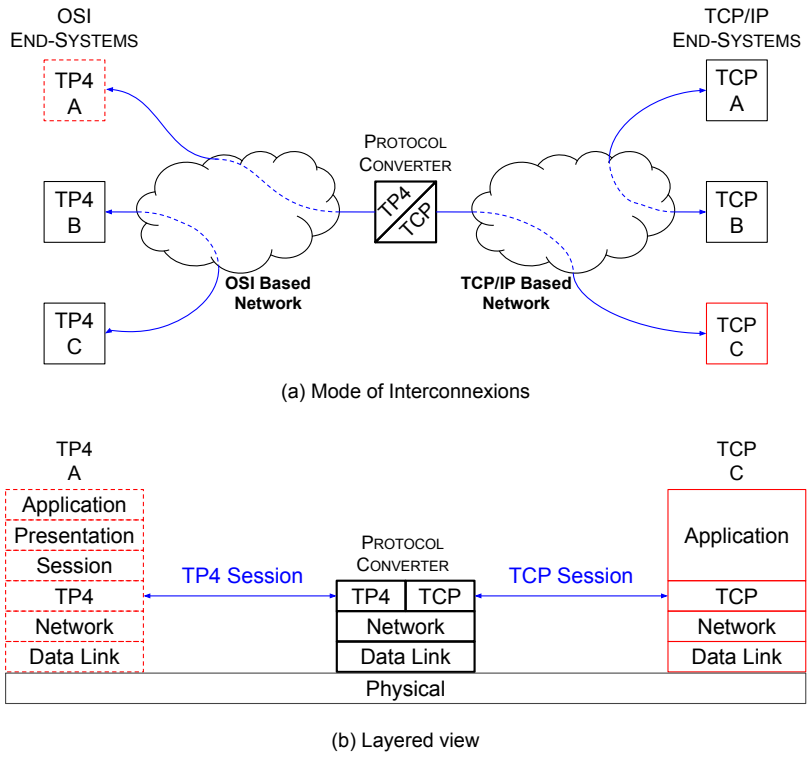


Figure 4.1: Protocols converter for OSI and TCP/IP end-systems. Adapted from [72].

be useful to address the limited use of Transport protocols. To achieve this goal, the use of the proxy should not require the modification of the applications on the end systems, i.e. the client application should not have to send data packets to the proxy but instead continues to send data to the server application: in this case, the proxy is qualified as a *transparent proxy*. Following this approach, several works have aimed to stimulate the use of Transport protocols other than TCP, in particular SCTP or MPTCP. In order to gradually enable the deployment and the use of SCTP on the Internet, the authors of [77] and [78] propose TCP-SCTP mapping system for transparent redirection of TCP connections to SCTP. In [77], the mapping tool acts like a transparent proxy called CMG (Connection Manager Gateway) that merges multiple TCP connections into a single SCTP association whereas in [78] the mapping tool is a "shim layer" designed to be directly integrated into the end-system OS. Similar to CMG and Shim Layer, MiMBox [79] is a protocol converter that ensures the translation between the regular TCP and its multipath extension i.e. MPTCP.

However, the above-mentioned solutions present two main drawbacks. First, they address only the adoption issues of only one Transport protocol: for instance, SCTP in the case of CMG and Shim Layer mapping tools, and MPTCP in the case of MiMBox protocol converter. They are what we could call a *one-to-one* protocol translator and therefore do not provide a comprehensive way for mapping TCP to multiple protocols. Second, even if there is no need to alter the application itself, most of those solutions require the change

Table 4.1: Primary functions of (TCP) socket API.

Functions	Parameters	Description
`socket()`	`domain, type, protocol`	Create a new socket.
`bind()`	`sock_fd, addr, addr_size`	Assign a local IP address and port number to a socket.
`listen()`	`sock_fd, max_connection`	Set the socket in listen mode.
`connect()`	`sock_fd, addr, addr_size`	Make a connection request and bind the socket to the remote IP address and port number.
`accept()`	`sock_fd, addr, addr_size`	Accept incoming connection request.
`send()`	`sock_fd, data, data_size`	Send data over a socket.
`recv()`	`sock_fd, data, data_size`	Receive data from a socket.
`close()`	`sock_fd`	Teardown a connection.

of the socket API thanks either to kernel patches such as done by Shim Layer, or to the preloading[1] technique like in CMG. Further, MiMBox is developed as a Linux kernel module and as such, it inherits the drawbacks associated with kernel modules namely the lack of security and safety of the end-system (see Chapter 3).

1: Conceptually, the preloading technique consists of overloading functions namely system calls of the OS kernel [80]. It relies on the LD_PRELOAD environment variable.

Contribution positioning. The approach we explored in this contribution allows the invocation, during the execution of the legacy application, of the alternative protocol *X*, without any modification of the application's code. This approach, which we introduced and implemented through the *Hooker* VTL component, leads at the level of the host machines to intercept the system calls related to the socket API (i.e., `connect()`, `sendmsg()`, `recvmsg()`, etc.) in order to ultimately invoke the alternative protocol *X*. Therefore, the *Hooker* component does not act as a simple proxy insofar as *(i)* the TCP protocol is not activated but rather replaced by the protocol *X* and *(ii)* there is ***no*** one-to-one static and permanent mapping of TCP to a single Transport protocol.

4.2 Socket API Layer and TCP Execution Path

A *socket* is a special structure that provides an abstraction of a bidirectional endpoint represented by the tuple IP address and port number. Technically, it is identified by a file descriptor, i.e. a small non-negative integer. The *socket API* is a set of standard functions that are used to manipulate *sockets* in order to correctly send and receive data. The commonly used functions of the socket interface are provided in Table 4.1. Those functions fail in two main categories: *(1)* the configuration functions used for instance to create and to set up a socket, to establish or to close a connection between different sockets, etc.; and *(2)* the I/O functions namely `send()`

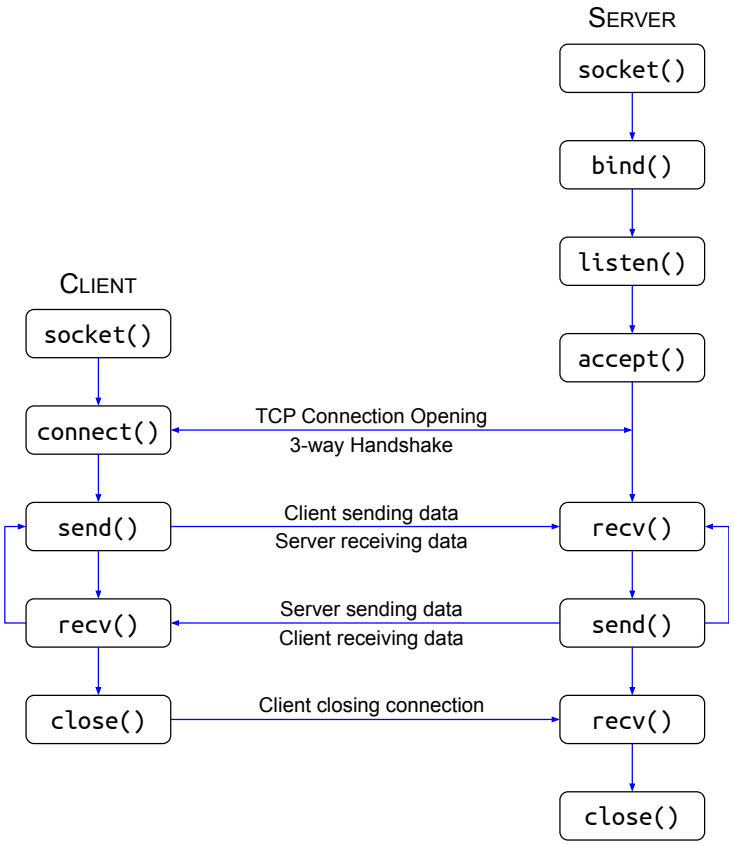


Figure 4.2: A basic sequence of socket API function calls during data transfer.

and `recv()` that are used to transfer data between applications. Fig. 4.2 illustrates a typical sequence of function calls between TCP applications that use the socket API during a data transfer.

On the client-side as well as on the server-side, the application uses `socket()` function to create a new socket. The `socket()` function enables the application to specify the domain (also known as protocol family) that should be attributed to the socket. Possible values of the domain parameter include but are not limited to: `AF_INET` (resp. `AF_INET6`) for IPv4 protocols family (resp. for IPv6 protocols domain), `AF_UNIX` for local communication domain, or `AF_XDP` for XDP interface family. For a TCP application, the *type* parameter value is always `SOCK_STREAM` that indicates that the socket should provide connection-oriented, byte-stream-oriented, bidirectional, and reliable Transport services. Recall that the socket API imposes on applications to explicitly specify the L4 protocol to be used to transfer data. That is the role of the *protocol* parameter of the `socket()` function. If this latter parameter is set to 0, the default L4 protocol associated with the *type* parameter is used: it is TCP protocol in the case of `SOCK_STREAM`.

Since the socket returned by the `socket()` function is a simple file descriptor that is just an integer, to be ready to send and to receive data through this socket, the TCP application needs to associate with the socket a tuple composed by one IP address

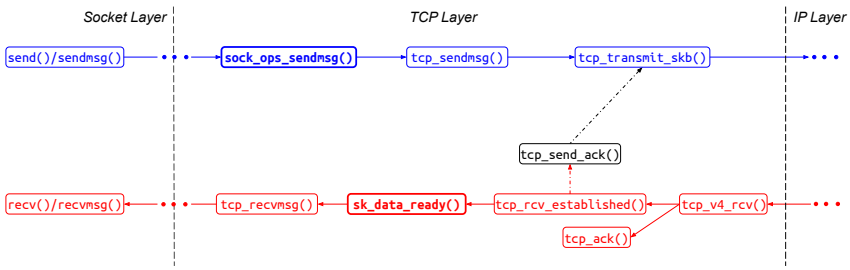


Figure 4.3: Simplified TCP Execution Path on egress and ingress paths. Full version is available at appendix A.

and one port number. To do so, on the client-side, the application should use `connect()` function to make a connection towards the remote server and bind the remote IP address and port number to the socket in case of a successful connection. To do the same, the server will call sequentially `bind()`, `listen()` and `accept()` functions. Now, the client and the server could enter in their data transfer loop and use `send()` (resp. `recv()`) to send data (resp. to receive data). After data transfer completion, the socket is closed thanks to the function `close()`. The closing is necessary to free resources such as the file associated with the socket or the IP addresses assigned to the socket during connection opening.

The invocation of each function of the socket API will trigger a call to further functions on the TCP execution path necessary to convey data packets. Let us assume that the TCP connection is established and let us take a look at the packet journey at the sending and the receiving of data packets from the socket API function call to the end of TCP's network kernel path and vice versa (see Fig. 4.3). When the application invokes `send()` function to transmit data, the socket layer, after completion of its own operations, uses the virtual function `sock_ops_sendmsg()` to pass the control to `tcp_sendmsg()` kernel function that first reserves memory space necessary to store the application payload as well as its associated protocols' headers. In Linux OS, for instance, this memory is the so-called `skb_buff` (see note 5 in Chapter 3). After the memory allocation, `tcp_sendmsg()` copies application data into the reserved memory and passes the control to `tcp_push_one()` function. Again, TCP is a reliable protocol that is based on the retransmission of lost packets. As such, the sending TCP entity should save a copy (more precisely a clone) of the packet and trigger a retransmission timer. This is the task of `tcp_push_one()` function which puts the packet into the retransmission queue and associates a timer to that packet. Finally, before passing the packet to the IP layer, the sending TCP entity builds a complete TCP packet with the help of `tcp_transmit_skb()` function that adds the TCP headers to the packet.

At the reception of data packets, once the control is passed to the `tcp_v4_rcv()` function, it dispatches the packet either to the `tcp_ack()` or to the `tcp_rcv_established()` function. Indeed,

when the received packet is an acknowledgment packet, `tcp_ack()` deletes the acknowledged packet from the retransmission queue and stops the associated timer. The received packet that contains data for the application is passed to the latter thanks to the `tcp_rcv_established()` function. If the application process is already waiting for data, the packet is directly copied to the user-space memory, otherwise, the `sk_data_ready()` virtual function is used to wake-up the user application and signals to it that new data are available for reading.

The insight of the TCP layer and socket API functioning and interactions is necessary to make the most appropriate design and technical choices toward *Hooker* component implementation. Indeed, each of the above-described functions could be used as a "hook" point, i.e. and for recall, the invocation of the hooked function will systematically trigger the execution of an attached eBPF program. For instance, an eBPF program could be attached to `connect()` function in a way that each time TCP application tries to establish connection it can be identified and registered if necessary. Also, eBPF programs could be attached to `tcp_transmit_skb()` kernel function so that the latter add a custom TCP option during headers forming [81].

4.3 Hooker Design and Implementation

Hooker goals and requirements. *Hooker* is the component of VTL that achieves the dynamic and transparent replacement of TCP by another protocol *X*. To achieve this objective, the *Hooker* component must interrupt TCP's execution path (described in the previous section). At data sending, as soon as the application calls into the `send()/sendmsg()` function, *Hooker* must take control of the packets before the kernel network stack. Therefore, it is necessary to place a hook point on the `tcp_sendmsg()` function so that each time this latter function is invoked, the *Hooker* component executes a dedicated program before the kernel. As for incoming packets, they should be intercepted as soon as they arrive at the network interface card (NIC), here also, to avoid their control by the kernel network stack. The conceptual and technical choices we made to meet these different specifications are described in the following paragraphs and subsections.

Functional Architecture Overview. Resulting from the above requirements, Fig. 4.4 depicts the internal structure of *Hooker* component and its interactions with the kernel network stack as well as with the legacy applications. Conceptually, *Hooker* is separated in three main subcomponents: *hooker_userspace*, *hooker_egress*, and *hooker_ingress*. As its name suggested, *hooker_userspace* is a normal

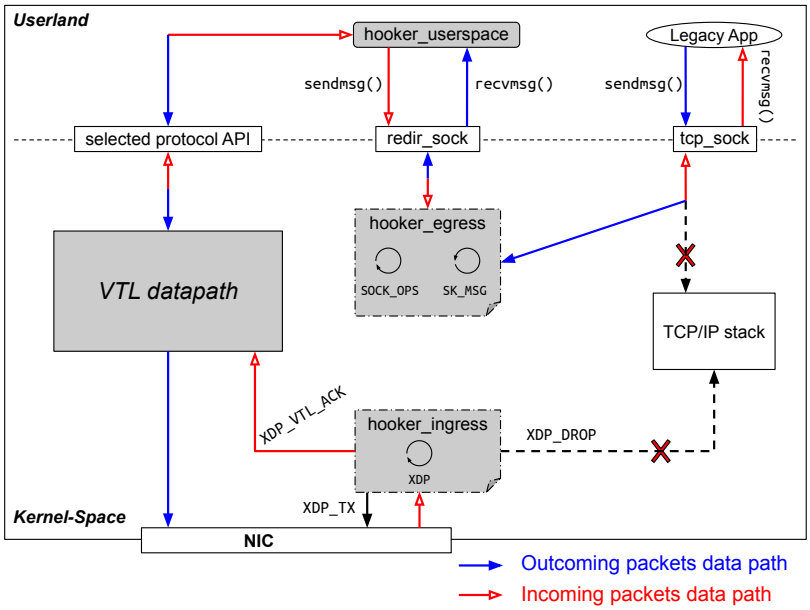


Figure 4.4: VTL *Hooker* Component Internal Structure.

program running in user-space and that, among other tasks, is in charge of creating and configuring sockets, namely the redirection socket (`redir_sock`) and the socket of the selected Transport protocol *X*. The rest of *Hooker's* subcomponents, i.e. *hooker_ingress* and *hooker_egress*, are eBPF programs and as such, they are executed in the kernel-space and deployed by *Launcher* component of the VTL (go back to Fig. 3.3 of Chapter 3) as soon as *Hooker* is activated. The eBPF program composing *hooker_ingress* is an `XDP` program that attaches to the network interface driver in order to process as early as possible all incoming data packets. The previous chapter describes in detail the `XDP` program and its associated hook. Hence, in the next sections, we will principally present the different types of eBPF programs as well as their associated hooks that constituted *hooker_egress* subcomponent.

Design choices discussion. In the architecture illustrated in Fig. 4.4, the choice we made was to pass the *hooker_userspace* subcomponent in the user-space. This choice, which initially meets a proof-of-concept purpose, opens up in a more global perspective the possibility of using protocols deployed both in kernel-space and in user-space such as QUIC (we will see it in the next chapter). At the price of higher implementation complexity, it is quite conceivable to bring the *hooker_userspace* subcomponent back into kernel-space. Our current hypothesis is that this could improve performance (which however, as we will see later, remains at an acceptable level with the current implementation choice of leaving the *hooker_userspace* subcomponent in user-space). Though, pushing back the *hooker_userspace* in kernel-space will deprive us of the use of user-space protocols like QUIC for which, ***no*** kernel implementation exists as of this thesis writing.

4.3.1 SOCKMAP: the art of data packets stealing

To redirect data between the TCP socket (`tcp_sock` in Fig. 4.4) and the redirection socket (`redir_sock` in Fig. 4.4), the most straightforward (but bad) idea would be to use the classical `send()/recv()` approach in which the TCP application sends the data to the *hooker_userspace* via the address attached to the redirection socket (`redir_sock`) and vice versa. This approach is illustrated in Fig. 4.5. Despite its simplicity, the `send()/recv()` method presents two drawbacks. First, significant overheads might be generated due to the fact data will traverse twice the TCP kernel path at the outgoing as well as at the incoming of data packets. Second, the legacy application should be rewritten to send data to the local redirection socket rather than to the remote server that is the default destination. By doing so, this solution does not fulfill the *transparency* requirement. The above-described `send()/recv()` approach is also known as a non-transparent proxy practical implementation inside an end-system.

Without any modification of the legacy application as well as the TCP's execution path (shown in Fig. 4.3), once the application issues a `send()/sendmsg()` function call, the kernel systematically takes control, handles the data encapsulation, and sends encapsulated data packets over the network directly to the remote server without passing by the *hooker_userspace* subcomponent. Herewith, the main concern is, when the application calls send()/sendmsg() function upon the TCP socket, *how to get data before they reach the kernel network stack in order to interrupt TCP execution path, and then to redirect the data towards another L4 protocol*? As discussed in Chapter 3 and illustrated in Fig. 4.7, `XDP` and `TC` hooks cannot take control of the data packets before the kernel Transport layer, i.e. TCP kernel processing in case of TCP applications. Fortunately, `SOCKMAP` and its associated `SK_MSG` eBPF program could be used to comply with the transparency requirement and at the same time to pick up the data before the kernel network stack. Besides transparency, this approach permits also to reduce the overheads introduced on the kernel's network path.

`SOCKMAP` is a *special* type of *eBPF map* and as such, it stores a set of *{key, value}* tuples. The specificity of this map is that the *value* at each index of the map can only be a TCP socket descriptor (see Fig. 4.8). A `SOCKMAP` has attached to it, at least, one eBPF program that gets executed upon data receipt on one of the TCP socket descriptors stored in the map. Under *Hooker*, it is an `SK_MSG` eBPF program that is part of *hooker_egress* subcomponent. `SK_MSG` is a type of eBPF program that gets executed before the `tcp_sendmsg()` function upon a `send()/sendmsg()` call on *any* TCP socket *stored* in the `SOCKMAP`. By taking the control of the data packets directly from

For recall, maps are generic data structures storing a set of key, value pairs used to exchange data either between different programs distributed between user-space and kernel-space.

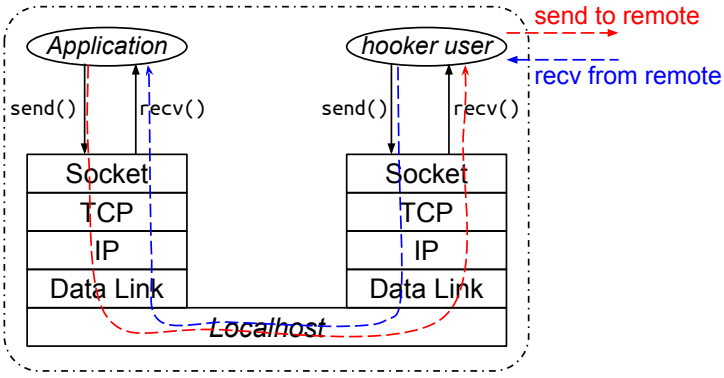


Figure 4.5: Traditional data redirection between sockets on the same host. The kernel network path is crossed twice at the sending as well as at the receiving of data.

the socket layer before TCP kernel network stack, `SK_MSG` could, for instance, seamlessly redirect the grasped data towards another socket (e.g. redirection socket in the case of *Hooker*). `SOCKMAP` data redirecting between sockets is illustrated in Fig. 4.6.

4.3.2 `SOCK_OPS`: TCP Execution Path's Spy

Once a `SK_MSG` program is attached to a `SOCKMAP`, it also gets systematically attached to all TCP socket descriptors stored in the map. In this way, the `SK_MSG` program is executed each time a write operation is performed on any one of the stored socket descriptors. Hence, the `SOCKMAP` needs to be priorly populated with the right values of these TCP socket descriptors. However, the value of a TCP socket descriptor of the legacy application is unknown before the end of the three-way handshake. Indeed, this value is attributed randomly by the kernel at the end of the TCP connection establishment. As a consequence, there is a need to accurately monitor the TCP's execution path in order to get notified at the end of a passive as well as active connection. The aim is to grab and add the value of the attributed socket descriptor in the `SOCKMAP` just before the application issued the first `send()/sendmsg()` call. To fulfill this goal, we used the `SOCK_OPS` eBPF program.

`SOCK_OPS`, introduced in the TCP-BPF framework [82], is a new

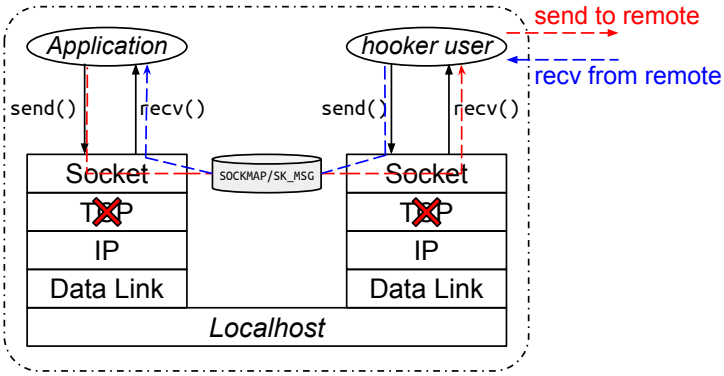


Figure 4.6: Data redirection between sockets based on `SOCKMAP` and its associated `SK_MSG` eBPF program. The kernel network path is bypassed and its subsequent overheads is eliminated.

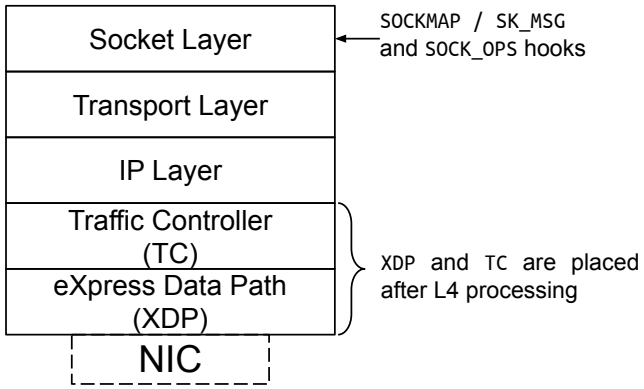


Figure 4.7: SOCKMAP/SK_MSG and SOCK_OPS eBPF hooks location in the network stack.

type of eBPF program that allows getting notified upon a call to any function of TCP's execution path. Originally, it was released to enable the programmability of the TCP layer by the means of fine-tuning of TCP's connections parameters from userland application processes. For instance, SOCK_OPS program could be used to dynamically set up the value of the initial congestion window (INIT_CWND) of a TCP connection or the buffer sizes of a TCP socket. SOCK_OPS programs rely on the single system call `tcp_call_bpf(..., int op, ...)` that could be called from any function on the TCP's execution path. The parameter op is used to signal to the eBPF program exactly from which function of TCP execution path the system call is invoked. This is a key feature that permits accurate monitoring of TCP sessions from connection opening to the closing of the connection without a need to add any code to the legacy applications.

A selected list of possible values of *op* parameters is shown in Table 4.2. For instance, when the *op* parameter value is BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB, the eBPF program knows that the call comes from the `tcp_finish_connect()` function and can then deduce that an active connection requested by the application has been successfully established. After being notified about some TCP events such as connection establishment, the eBPF program can, for

op	*Description*
TCP_CONNECT_CB	Call eBPF program right before an active connection is initialized.
ACTIVE_ESTABLISHED_CB	Call eBPF program when an active connection is established.
PASSIVE_ESTABLISHED_CB	Call eBPF program when a passive connection is established.
TCP_LISTEN_CB	Call eBPF program on listen() invocation, right after socket transition to LISTEN state.
RTT_CB	Call eBPF program on every RTT.

Table 4.2: List of selected eBPF SOCK_OPS operators (*op*) in Linux 5.3.5. The name of each *op* must be preceded by the prefix BPF_SOCK_OPS

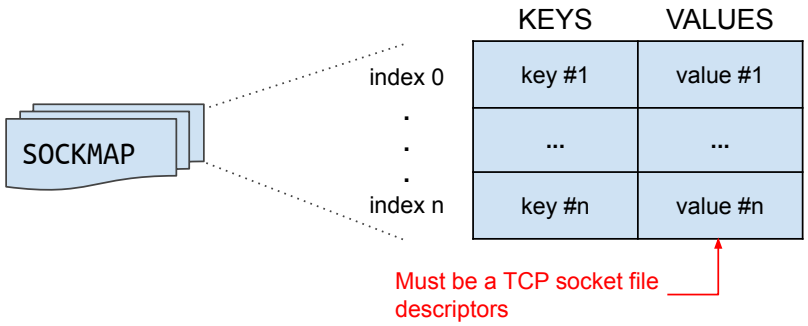


Figure 4.8: SOCKMAP illustration. The map value can only be a file descriptor of listening TCP sockets.

instance, decide to register the TCP socket descriptor attributed to the application in the special eBPF map SOCKMAP. From that point, every data from the application could be seamlessly grasped and redirected during data transfer. Within the *Hooker* component, we used a SOCK_OPS program as a part *hooker_egress* subcomponent in order to identify, register, and monitor TCP applications of interest based on their port number. This is only an implementation choice; one can easily decide to blindly handle and monitor all TCP applications within the end-system OS.

4.3.3 Legacy Application Data Paths

The internal structure of *Hooker* component (see Fig. 4.4) also depicted the application data packets paths from the send()/sendmsg() call to the transmission over the network interface card and vice versa. Once it is activated, *Hooker* attaches three different types of eBPF programs at various levels of the network stack: *(1)* a SOCK_OPS program attached to the *root* cgroupv2 [83], *(2)* a SK_MSG attached to a SOCKMAP at the socket layer, and *(3)* an XDP program placed at the network interface card (NIC) in order to process incoming data packets. By leveraging the hierarchical model of cgroups, *Hooker* is able to process at the socket layer any ingress and egress data packets of all TCP application processes running on the end-system.

Hooker maintains several maps, especially the SOCKMAP described below. In the first implementation of *Hooker*, the key at each index of the SOCKMAP is a structure containing the addressing information enumerated in the code Listing 1. In order to optimize the map manipulation operations such (updating, searching, etc.), the key in the current version of the *Hooker* component is a small integer that is a hash of the addressing information listed in the code Listing 2. This key is used by the *hooker_egress* programs to identify the right socket towards which the packet data must be forwarded to. Furthermore, as previously discussed, the SOCKMAP is helpful to keep a trace of applications whose packet data should be intercepted and redirected by *Hooker*. Each time a connection

Listing 4.1: Example of a SOCKMAP key structure.

```
struct sock_key {
        __u32 src_ip4;
        __u32 dst_ip4;
        __u32 src_port;
        __u32 dst_port;
}
```

is established or closed by one process, the map is updated by *hooker_egress* thanks to the `SOCKS_OPS` bpf program section attached to `cgroupv2`. In addition to `SOCKMAP` updating at the connection opening, the `SOCK_OPS` bpf section is used to add to the SYN packet a VTL_COMPLIANT option that, as its name suggested, is useful to advertise to the receiver that the sender is VTL compliant. This feature is more described in Chapter 5.

Every time the TCP application process sends data by the invocation of the `sendmsg()` function upon the TCP socket, the `SK_MSG` bpf program running by *hooker_egress* intercepts the data packet and rewrites it if necessary thanks to the helper function `bpf_msg_push_data()`. Then, to redirect the egress data packets to the redirection socket, *hooker_egress* program leverages the `bpf_msg_redirect_map()` helper function. At the incoming of data packets, *hooker_egress* uses the same helper to redirect the packets to the TCP socket. 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 that should emulate the selected Transport protocol functioning. At the receipt of a data, as soon as the network interface driver (NIC) receives the data packet, the `XDP` bpf program section 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` verdict), redirect it to the same network interface card (`XDP_TX` verdict), or, as currently done, pass the packet to the ingress VTL datapath (`XDP_VTL_ACK` verdict) for further processing.

4.4 Performance Evaluation

Evaluation's goals. *Hooker* is evaluated through experiments based on the transfer of three files of different types (text, image, and video) having different sizes (64 KB, 512 KB, and 1 MB respectively). We assume that the transmitted multimedia (image and video) allows a data packets loss rate lower than 10%. We empirically validate this assumption by observing whether the received multimedia is clearly readable by human eyes. However, as stated in Chapter 3, more formal approaches to compensate data packet

VMs Specifications		
CPU	***RAM***	***NIC***
Intel 2.70GHz × 4	3.8 GiB	Atheros QCA6174
Emulated Network		
RTT	***Bandwidth***	***Loss rate***
100 ms	100 Mb/s	0-5%

Table 4.3: Tested network and hosts' Configurations.

	Compilation	*Deployment*	*Redirection ops*
SK_MSG	0.06 s	0.062 s	11 μs
SOCK_OPS	0.09 s	0.064 s	N.A
XDP	0.08 s	0.057 s	N.A
Hooker User	0.456 s	N.A	2019 μs
Total	**0.686 s**	**0.183 s**	**2030 μs**

Table 4.4: Data redirection cost and *Hooker* activation delay.

losses should rely on methods such as redundancy (e.g., FEC [84]) or adaptive coding [60]. The consideration of these methods is out of the scope of the goal of the experiments we carried out here. Indeed, the main goal of the carried out assessment is to demonstrate the ability of *Hooker* to effectively replace TCP with another L4 protocol during legacy applications data transfer. Therefore, we performed the evaluations by considering the above assumptions on the application requirements and with a limited set of alternative L4 protocols notably UDP and QUIC. In addition to the functional evaluations, we measured the cost in terms of delay of the data redirection operations, as well as the delay of the dynamic deployment of the *Hooker's* programs namely the SOCK_OPS, SK_MSG, and XDP programs. Finally, we evaluated each protocol's performance in terms of file completion time (FCT) that, as defined in Chapter 3, is the time necessary to complete the transfer of a whole file.

Testbed configurations. The topology of the testbed used during experiments is similar to the one used in Chapter 3 and is illustrated in Fig. 5.1 of the next chapter. For recall, the client host as well as the server host run on Linux 5.3.5 OS and are equipped with Intel Core i7-7500U CPUs and 3.8 GiB RAM. Here also, we used `netem` emulator to set the RTT of the network link to 100 ms, the link capacity to 100 Mb/s, and the loss rate between 0 and 5%. Table 4.3 provides a summary of the specifications of the VMs and the emulated network that constituted the experiments' environment.

Evaluation scenarios and validation approach. For each file, we considered the identical scenario consisting of two steps: *(1)* a first data streaming is performed without running *Hooker*, and *(2)* in a second time, we performed the data transfer under the activation of the *Hooker* component. In the first case, the *native* performance of the application on top of each Transport protocol is measured. During the second stage, we measured the performance in terms of file completion time (FCT) of the TCP application that accesses the alternative Transport protocol *X* thanks to the *Hooker* VTL component. At each step, we first check that the client correctly received the streamed file. Then, with the Wireshark analyzer [5], we validate the correctness of the data redirection by checking the Transport protocol used on the wire during data transfer.

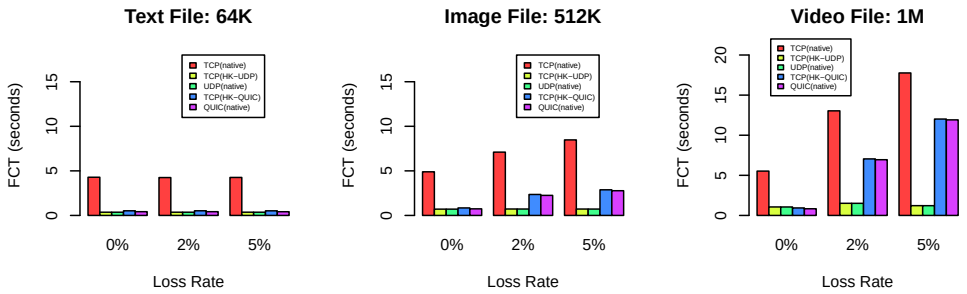


Figure 4.9: Latency of the data streaming with and without *Hooker*. TCP(HK-UDP) shows TCP application performance when Hooker replaces TCP with UDP during data transfer. The same note applied for TCP(HK-QUIC).

Microbenchmarks. As reported in Table 4.4, it takes less than one second to activate the *Hooker* component. Further, we could also note that when the *Hooker* is precompiled, its activation delay is reduced to less than 200 ms. Besides the activation delay, once the *Hooker* is activated, its operations namely data redirection introduce additional overheads. We computed these overheads in terms of the average delay required for data redirection operations that are achieved in *hooker_egress* (i.e. `SK_MSG`) and *hooker_userspace* subcomponents. The results are reported in Table 4.4 and showed that it takes approximately 2 ms to redirect packets during data transfer. Finally, the results reported in Fig. 4.9, characterized the achieved file completion time of each protocol in various network conditions. As we can note, natively, TCP applications take more time to complete the file transfer. This time is considerably reduced when *Hooker* seamlessly replaces TCP either by QUIC or by UDP during data transfer. Note that contrary to QUIC, if the alternative protocol is UDP, there will be data packet losses when the network link loss rate is not null. This could be crippling if the application does not tolerate losses. However, if the application tolerates losses, UDP looks like the best alternative since it takes to it less time in any network situation to complete data transfer. The choice of the better alternative is a tradeoff between the application requirements and the network conditions. The necessary algorithms to achieve this choice is the subject of the next Chapter.

4.5 Conclusion

In this chapter, we presented the design, the implementation, and the evaluation of the *Hooker* component. *Hooker* is the VTL component that enables the transparent integration of legacy applications into the VTL system by achieving the replacement of TCP with any other Transport protocol during application data transfer. The *transparency* property refers to the fact that the dynamic replacement of TCP is realized without any modification of the legacy application. This is a key factor to ease and to stimulate

the use of VTL as well as its associated Transport protocols either deployed by VTL or already existing in the end-system OS.

We carried out thorough experiments to demonstrate the effectiveness of *Hooker* component, i.e. its ability to replace at runtime TCP by alternative Transport protocol *X* without any modification of the legacy application. Further, the performance evaluation of *Hooker* showed that once it is activated, its data redirection operations take approximately less than 2 ms to complete. Nevertheless, the performance in terms of file completion time (FCT) achieved by the legacy application on top of native TCP is significantly poor compared to the performance achieved by the same application when *Hooker* redirects its connection and data seamlessly towards alternative Transport protocols. That is to say, even if *Hooker* introduced additional overheads that impact the native alternative protocol *X*, it considerably improves the performances of its initial target, i.e. the TCP applications. Besides the transparent integration of legacy applications within the VTL, this performance improvement is an additional motivation to the design and implementation of the *Hooker* component.

Finally, the evaluation also showed that the best alternative Transport protocol to TCP is not always the same and varies depending on the network conditions and the legacy application requirements. In the next chapter, we will discuss the proposed algorithms and approach for the selection of the best alternative L4 protocol to the TCP.

Optimal Selection of Protocols | 5

In the previous chapter, we substantially presented our technique that allows legacy applications to use another Transport protocol other than TCP without code rewriting. Further, the experiments carried out during this preceding chapter also showed that there is ***no universal*** alternative L4 protocol to TCP. That is to say, the best Transport protocol varies with the application requirements and the network state. Even worse, if the alternative protocol *X* is chosen without caution, the application could present suboptimal performances compared to its initial performances under TCP, as we will see in the first section of this chapter. Consequently, it sounds essential that we have an approach that enables VTL to craftily select the alternative to TCP based on the application requirements and the network conditions. In this chapter, we propose such a method that ensures selecting the best choice to replace TCP during data transfer.

To select the most appropriate L4 protocol to the network context and the application requirements, we used machine learning models, namely decision trees, that we trained using the C5.0 algorithm (presented later in this chapter). Since application QoS requirements and network conditions are the decision tree models' attributes, we need first to get both information. However, a legacy application does not express its QoS requirements via the standard socket API. Indeed, this API's invocation simply expresses the desire to see the TCP (or UDP) protocol activated. All we know when hooking on some legacy application is that it is a *TCP-based application*. The critical concern is then *how to get the hooked TCP application QoS needs*? To cope with these concerns, we proposed *(i)* a profiling technique enabling us to infer the requirements of the legacy applications, and *(ii)* a parsimonious[1] network monitoring useful to estimate the state of the network in terms of RTT, maximum link capacity, and packet loss rate. On this basis, the decision tree models, feeding the VTL knowledge base, could be used to select the "best" L4 protocol.

1: Parsimonious monitoring refers to the fact that monitoring operations have little or no impact on the data traffic load.

To train the decision trees, we generated a dataset via thorough evaluations of several IETF L4 protocols under various network conditions and different application requirements. Part of these evaluations is presented in Section 5.1 as a preamble of this chapter to motivate the need for an approach in order to carefully select the alternative protocol to TCP during data transfer. In Section 5.2, we provide the theoretical background on the

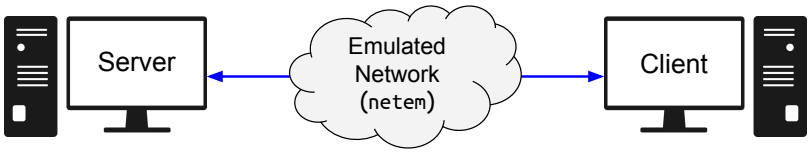


Figure 5.1: The considered point-to-point topology for experiments. The link between Client and Server emulates either a classical terrestrial Internet or a satellite Internet link.

machine learning techniques, namely decision tree classifiers and the C5.0 algorithm that we used to induce the decision trees. Section 5.3 presents the design and implementation of the proposed approach. In particular, we describe the application profiling method (Subsection 5.3.1) as well as the network state estimation approach (Subsection 5.3.2). We end the section by detailing the application of the decision trees induction algorithm to our approach (Subsection 5.3.3). In Section 5.4, we focus on the evaluation of the proposed solution. We first assess the precisions and recalls of the constructed models. Then, we estimate how much the proposed selection approach enhances the performance (in terms of throughput) of TCP applications under VTL. Finally, we conclude the chapter in Section 5.5.

5.1 Motivation

Let us extend the experiments of Chapter 4 to another network type namely the satellite network links. Fig. 5.1 presents the point-to-point topology used for the experiments. Here, the link between the two hosts emulates either the classical terrestrial Internet or a satellite link. Section 5.4 describes in detail the network links configuration and emulation tools as well as the values of the network parameters (delay, bandwidth, and loss rate). The *Hooker* component is also activated. The performance criterion under observation is the throughput, whose formal definition has been given in Chapter 4.

Let us first consider the satellite link. Without packet loss, TCP has equivalent performance to QUIC as well as to Hybla[2] that is more adapted to satellite links [69]. However, once the link starts by experiencing losses, we notice a significant TCP throughput degradation. As it can be deduced from the results reported in Fig. 5.2, an application using TCP could, on average, get 3~4x better throughput on a satellite link if it used Hybla instead. Now, assume that in this context, instead of using Hybla, we selected QUIC. The results (Fig. 5.2 (a)) show that the performance is not only suboptimal compared to Hybla, but also, and more importantly, that it is worse than the initial performance of the application under TCP. In simple terms, QUIC will perform worse than TCP on the satellite link, when data packet losses occur.

2: Recall throughout this manuscript we used interchangeably TCP and TCP Cubic (the default version of TCP in the mainstream operating systems namely Linux). Hybla is an extension to TCP and as previously discussed, the alternatives to TCP as well as its own extensions suffer from the same issue: the lack of wide deployment and/or the limited use by applications on the Internet.

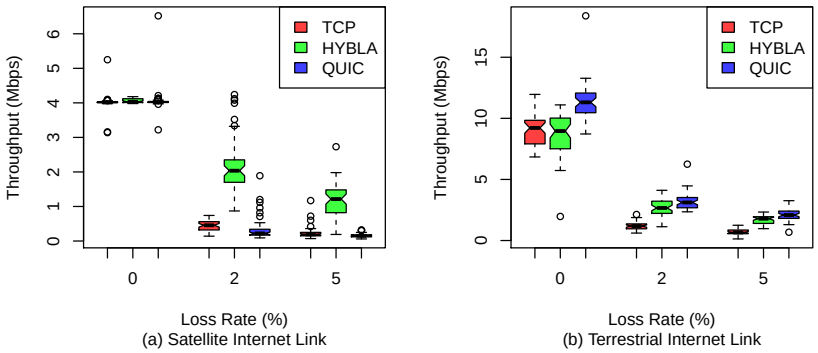


Figure 5.2: L4 protocols' performances under various network conditions. Boxes span the 25th to 75th percentiles, with a notch at the median.

From these first results, we could speculate that it would be enough to use Hybla continually as the alternative to TCP. Nevertheless, let us resume the same experiments on a terrestrial link. Here again, without data packet losses, TCP remains near equivalent to the QUIC and Hybla protocols. However, as soon as the network link suffers its first packet losses, a TCP application that would switch to QUIC will achieve about 1.5~2x better performance in terms of experienced throughput. Further, the results (Fig. 5.2 (b)) also demonstrated that contrary to the satellite link, QUIC presents better performance than Hybla and seems the best alternative to TCP in the terrestrial network context.

The more we continue the experiments by changing the state of the network and the requirements of the application, the more we observe that the best alternative to TCP changes regularly. This preliminary assessment allows us to validate TCP's performance limitations in specific environments, and demonstrates that replacing TCP may or may not be justified depending on the context. Hence, the interest of having an approach that permits VTL to choose the appropriate protocol X because, as we have just seen in the above experiments, the protocol X might not be the same in all network and application contexts and could even perform worse than TCP.

5.2 Background

For the sake of clarity, we describe here the theoretical background on machine learning models and algorithms, namely the decision tree classifiers and the C5.0 algorithm above which we built our approach. Traditionally, machine learning is recognized as the field that gives any computer system the ability to learn to do without being explicitly programmed for [85]. In other words, with the help of machine learning models *learned* most of the time off-line from past experiences (a.k.a. *datasets*), the computer system can take or predict future decisions alone. Roughly, the models could be trained following two principal learning approaches:

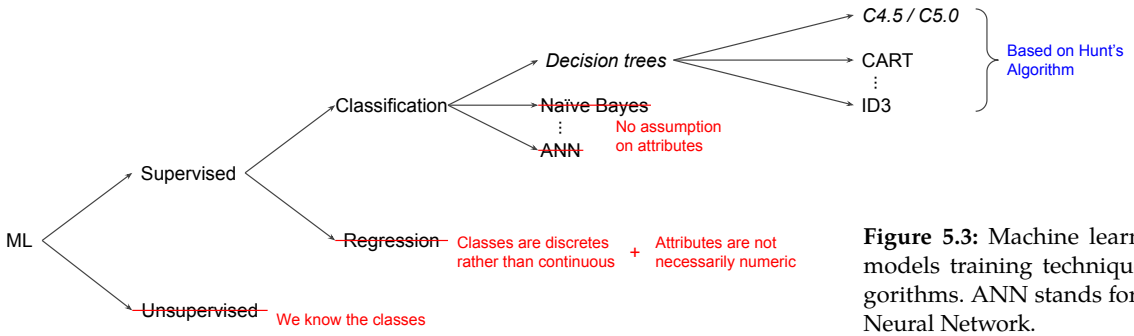


Figure 5.3: Machine learning (ML) models training techniques and algorithms. ANN stands for Artificial Neural Network.

(1) supervised learning and *(2) unsupervised learning*. In supervised learning, each case in the training dataset is associated with a class or label (i.e., the expected output). The dataset is a so-called *labeled dataset*. For instance, in our case, each case, i.e. the pair *{application requirements / network context}*, is associated with an L4 protocol from the set of the considered IETF Transport protocols. Unsupervised learning allows training models when the dataset cases' classes are unknown, which means the dataset is not labeled. Fig. 5.3 summarizes those learning techniques and justifies our use of decision tree models in this work. As the reader can guess, the models we used in our work are trained using the *supervised learning* technique.

5.2.1 Decision Tree Models

Suppose that we want to decide whether it is an excellent day to practice some activity outside based on the weather forecast information. The weather is characterized by four attributes: *outlook, temperature, humidity*, and *windy*. Table 5.1 provides a small *training dataset* of the decision taken in the previous 14 days based on those days' weather attributes. *Shall we play outside the day D15*? Intuitively, to predict the answer, one could use a sequence of questions on the values of the attributes of the day D15 and learn from the past decisions to make the right decision. The first question might be what the value of the *outlook* is? If the *outlook* is "overcast", we know from the *training dataset* that the decision will systematically be "yes" whatever are the values of others attributes: the series of questions ends. However, the day D15, the *outlook* is "sunny" rather than "overcast". We need to ask a second question about another attribute. Assume that the second question is on the level of *humidity*? If it is "high" (the value of the *humidity* level the day D15), we could deduce from the training set that the decision should be "no" and stop the sequence of questions.

This elementary example shows how, based on a series of questions, one could use the current measures and the background

	Outlook	*Temperature*	*Humidity*	*Windy*	***Decision***
D1	sunny	hot	high	false	no
D2	sunny	hot	high	true	no
D3	sunny	mild	high	false	no
D4	sunny	cool	normal	false	yes
D5	sunny	mild	normal	true	yes
D6	overcast	hot	high	false	yes
D7	overcast	cool	normal	true	yes
D8	overcast	mild	high	true	yes
D9	overcast	hot	normal	false	yes
D10	rain	mild	high	false	yes
D11	rain	cool	normal	false	yes
D12	rain	cool	normal	true	no
D13	rain	mild	normal	false	yes
D14	rain	mild	high	true	no
D15	sunny	cool	high	true	?

Table 5.1: A small training dataset from [86].

information from the past to predict or classify the future. A *decision tree* could be used to express this sequence of questions and their associated answers. In theory, a decision tree is a hierarchical data structure composed of nodes or vertices connected by a set of edges. A decision tree has two main types of nodes: *(1) leaf* nodes corresponding to a decision called *class* in decision theory terminology, and *(2) test* nodes that contain a *test condition* on the value of a specific attribute. Test nodes are either internal nodes or the root node of the decision tree.

5.2.2 C5.0 Algorithm for Decision Trees Induction

The *representation* of a decision tree is often straightforward, as well as its *use* for prediction and classification. However, at each stage, choosing the attribute on which the test condition should be applied is tricky. Let us go back to the previous example of subsection 5.2.1. If either at the first question, the choice was made to test the *windy* attribute or, after the first question on the *outlook* attribute and its associated "sunny" answer, the second question examined the *windy* attribute's value rather than the *humidity* attribute. In either case, the result would have been either a larger decision tree or a decision tree that is not only large but also inaccurate in terms of the quality of prediction and classification. For a single dataset, there exist a huge number of candidate decision trees. This number is exponential and, for instance, is greater than $4x10^6$ for the small training dataset of Table 5.1 [87]. As a result, contrary to its representation and its use, the *induction/construction* of an optimal and consistent decision tree is an NP-complete problem [87]. The academic literature is

plenty of different machine learning algorithms to find an *efficient* decision tree within an "acceptable" computational time. Many of those algorithms are fundamentally based on the recursive Hunt's algorithm (Fig. 5.3). As the algorithm's name suggested, it is proposed by Hunt et al. in 1966 [86] and can be summarized as follows.

Let us consider T as the training dataset consisting of past cases, and let us note $C = \{C_1, C_2, ..., C_n\}$, the classes. The recursive procedure is:

- T is not empty, and all the cases in T belong to a single class C_j: the decision tree for T is a leaf labeling C_j.
- T is empty: the decision tree is a leaf node labeled following some specific arbitrary rule. For instance, in the C4.5 variant, the most frequent class at this node's parent is attributed.
- T contains cases that belong to *more* than one class: find the "best" attribute on which the set T should be appropriately partitioned into smaller subsets. The best attribute is one that ***maximizes some local measures***. For instance, in C4.5, this measure is based on information theory metrics.

Similar to the local optimum search algorithms such as gradient descent [88], Hunt's procedure is a greedy algorithm that uses a heuristic based on the *maximization of some local measure*. The criteria used to evaluate this local measure is the main difference between the variants decision tree algorithms of Hunt's procedure. C5.0 is one of these algorithms. It heavily uses the information theory metrics as the criteria for the selection of the "best" attribute at stage 3 of Hunt's algorithm.

C5.0 is an improvement of the well-known C4.5 decision tree induction algorithm. As stated above, C5.0 is based on Hunt's algorithm. At the splitting step, i.e., stage 3 of Hunt's procedure, the criteria used by the C5.0 algorithm to select the "best" attribute are based on the information theory metrics, namely the *entropy* and the *information gains*. The main idea is as follows: the "best" attribute at each stage is one that provides the maximum *gain* of information. Let us consider T_i as any subset of the training dataset T, A as the attribute on which the splitting test is applied so that $T = \{T_1, T_2, ..., T_m\}$, and $C = \{C_1, C_2, ..., C_n\}$, as the set of classes. If $p(C_j, T_i)$ stands for the probability that all cases in the subset T_i belong to the class C_j, *entropy* and *information gains* are calculated by the following formulas.

$$p(C_j, T_i) = \frac{freq(C_j, T_i)}{|T_i|} \tag{5.1}$$

where:

- $freq(C_j, T_i)$ is the number of cases in S that belong to C_j,
- and $|T_i|$ is the size of the subset T_i.

$$Entropy(T) = -\sum_{j=1}^{n} p(C_j, T_i) \times \log_2(p(C_j, T_i)) \tag{5.2}$$

$$Entropy_A(T) = \sum_{i=1}^{m} \frac{|T_i|}{|T|} \times Entropy(T_i) \tag{5.3}$$

$$Gain(A) = Entropy(T) - Entropy_A(T) \tag{5.4}$$

where:

- $Entropy_A(T)$ is the weighted entropy of small subsets generated when splitting from attribute A.

$$Split\_entropy(A) = \sum_{i=1}^{m} \frac{|T_i|}{|T|} \times \log_2(\frac{|T_i|}{|T|}) \tag{5.5}$$

$$Gain\_ratio(A) = \frac{Gain(A)}{Split\_entropy(A)} \tag{5.6}$$

Basically, at each splitting step, C5.0 algorithm will compute the above values and will select the attribute that gives the highest gain ratio.

5.3 Protocols Selection Approach

We illustrate in Fig. 5.4 the high-level conceptual view of the approach adopted to select the most appropriate Transport protocol alternative to TCP. The selection relies on a set of decision tree models whose attributes are the application requirements and network conditions. The use of the decision tree models is therefore guided and preceded by *(1)* an application profiling that allows us to infer their needs and *(2)* a parsimonious monitoring that allows to estimate the current state of the network with limited/low cost on the link traffic. We discuss both techniques in this order in subsection 5.3.1 and subsection 5.3.2.

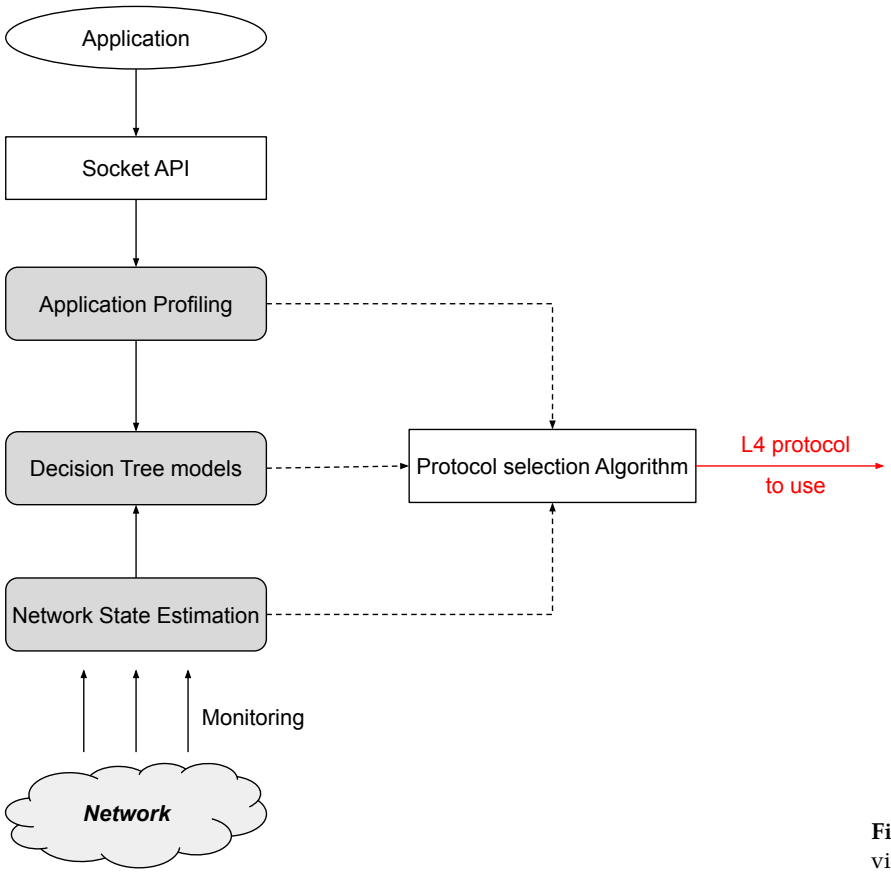


Figure 5.4: High-level conceptual view of protocols selection approach.

5.3.1 Receiver-driven Application Profiling

The purpose of profiling is to identify the nature of the TCP application. It permits to infer the requirements of the application. We have established the profiles on the basis of data from the ITU recommendations [89, 90]. To each profile, we associate requirements expressed in terms of Transport services and QoS parameters. The profiling, driven by the server (receiver of the connection), is initiated as soon as the first TCP packet (SYN) is received and continues over the following nine[3] packets. When profiling is successfully completed, the application is classified into one of the following profiles (see Fig. 5.5).

Profile 1: time-sensitive applications; e.g., multimedia streaming applications (YouTube, NetFlix, etc.) or videoconferencing applications (skype, zoom, etc.). Transport service requirements associated with this application profile are: partial reliability, partial order, and flow control to contribute into the jitter management. The multimedia streaming applications might tolerate a maximum delay of 10 seconds whereas the delay allowed by videoconferencing applications fluctuates between 10 milliseconds and several hundred milliseconds. The time-sensitive applications could experience a loss rate between 2 and 4%. However, it is worth noting that this profile of applications rarely uses TCP.

3: Profiling is attempted on the first ten packages. This number is arbitrary but higher than the recommendations in [91].

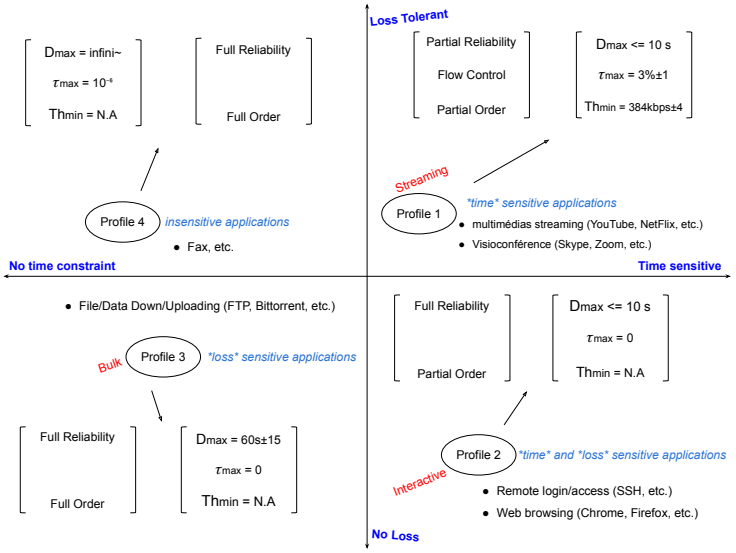


Figure 5.5: Applications profiles based on ITU recommendations. The associated requirements of each profile are also shown.

Profile 2: time and loss-sensitive applications; e.g., interactive applications such as remote login based on Telnet or SSH, web browsers (Chrome, Firefox, etc.), or online games (Call of Duty, Fortnite, etc.). The Transport services required by this application profile are: total reliability and partial order. The associated QoS parameters in terms of delay are a fraction of a second for remote login and online games whereas web browsing could accept delaying up to 10 seconds. The applications of this profile do not allow any loss of data.

Profile 3: loss-sensitive applications; e.g., (large) file transfer applications based on FTP/HTTP or BitTorrent, and email or text messaging / online chatting (Facebook Messenger, WhatsApp, etc.). For these applications, Transport services with total reliability and order are required. They do not tolerate any loss of data. On the other hand, these applications are less constraining with regard to the delay, which can go beyond 60 seconds.

Profile 4: insensitive applications; e.g., Fax. These applications are the least constraining in terms of packet loss and data transit delay.

The pipeline of application profiling by packet classification is shown in Fig. 5.6. The identification of the TCP application takes place in three main stages.

(1) Flow and IP packet extraction. In this work context, a flow is basically defined by the tuple $\{ip_{\text{src}}, ip_{\text{dst}}, port_{\text{src}}, port_{\text{dst}}\}$. The L4 protocol type information is not "necessary" because only TCP packets are processed, so it is impossible to differentiate flows based on this information. During this first step, matching between the intercepted raw packet and the flow table enables the extraction of the flow to which the packet belongs. If the packet does not belong to any stream in the table, a new stream is created. An IP

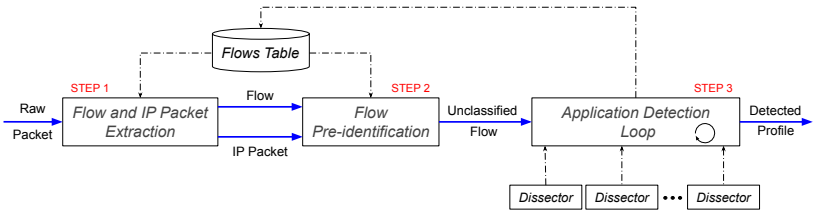


Figure 5.6: Application profiling pipeline.

packet is then extracted from the raw packet by removing the L2 header of the packet. This phase's final result is a flow and an IP packet ready to be used in the second phase.

(2) Flow pre-identification. The flows in the table might be already classified or not. A flow is classified when the application to which it belongs has already been detected. Therefore, the purpose of this phase is to directly retrieve this information from the table rather than systematically and blindly launch the application detection loop.

(3) Application detection loop. If the flow is not yet classified, it is either a new one or the first packets of the flow have not been sufficient to identify it. The detection loop is based on a hybrid approach to identify the application whose flow it receives: it integrates the signature-oriented approach based on protocol dissectors and the standard method based on port number mapping. It first attempts to identify the application by contrasting the flow to a set of predefined protocol dissectors. Dissectors are snippets of code that identify a specific protocol by reading/processing the entire IP packet (headers and payload included). For instance, an HTTP protocol dissector might fetch the "GET" string in the IP packet to determine whether the flow is an HTTP flow. As soon as a dissector correctly identifies the flow, the loop stops. If the application is not identified, the next packets of the stream (up to the 10th packet) are used to attempt a new detection of the application and its classification in one of the four profiles described above.

5.3.2 On-request Network Monitoring

In addition to the application's requirements, the network state is used to drive the selection of the best protocol X to replace TCP. To do this, we associate to each network link a state or profile characterized by three main parameters: $\{[RTT_{\min}, RTT_{\max}], BW_{\max}, loss_{\text{moy}}\}$. The $RTT_{\min}$ (resp. $RTT_{\max}$) denotes the minimum (resp. maximum) round-trip-time experienced under the network. The $loss_{\text{moy}}$ is the average rate of packet loss, and $BW_{\max}$ is the maximum bandwidth available within the network link. In Table 5.2, we can see that typical LDN networks such as satellite networks have profile P1 = $\{[500ms, \infty[, -, 1Mbps\}$ [92]. Note that the loss parameter

Table 5.2: Network profiles based on the link quality parameters.

	RTT	***Bandwidth (B.W)***
Long-delay Networks (LDN, e.g. Satellite)	>= 500 ms	4 Mbps
Terrestrial Internet Links	50 ms to 500 ms	100 Mbps
LAN (e.g. Internal D.C, home network)	<50 ms	100 Mbps to 100 Gbps

is neither static nor closely bound to a specific network profile but depends more on the network's congestion state. Therefore, it is possible (probably the fact) to experience more data packet losses under congested wired-LAN than non-congested wireless-LAN.

Since the bandwidth (the incoming data rate, in fact) estimation does not require any packet injection into the network, the monitoring component *continuously* captures a copy of the incoming packets to deduce the network link's bandwidth. However, we estimate the RTT and loss rate values by injecting out-of-band, albeit lightweight, ICMP ECHO/REPLY packets on the network. To minimize the impact of monitoring on the network traffic load, monitoring these two parameters is only triggered on demand through a set of functions exposed by the internal API of the monitoring component (*NetMon* in Fig. 3.3 of Chapter 3). The caller of the monitoring component has the possibility to specify the periodicity of the monitoring as well as its duration. The period defines the time interval between packets injection for calculation of RTT and loss rate. The larger the interval, the less expensive the monitoring is at the price of the estimation's accuracy.

5.3.3 Construction of Decision Tree Models for Protocols Selection

Application profiling and network monitoring are prerequisites to the selection of the most suitable alternative L4 protocol to TCP. They provide two information: the *application profile* (i.e., its requirements) and the *network state*. This information is the attributes (i.e., the inputs) of decision tree models on which are

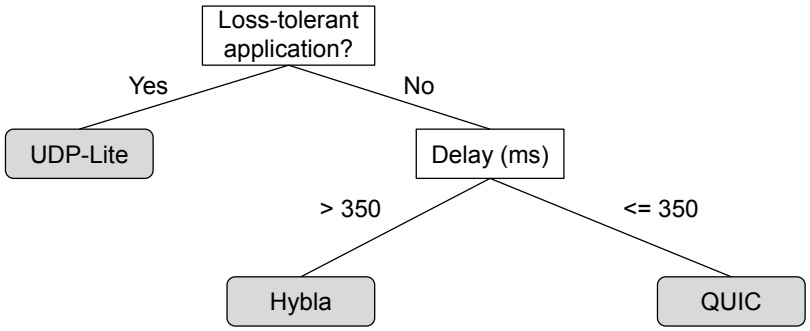


Figure 5.7: Simplified decision tree to select the most appropriate protocol. Leaf nodes (gray box) represent classes, whereas internal nodes (white box) represent the attributes.

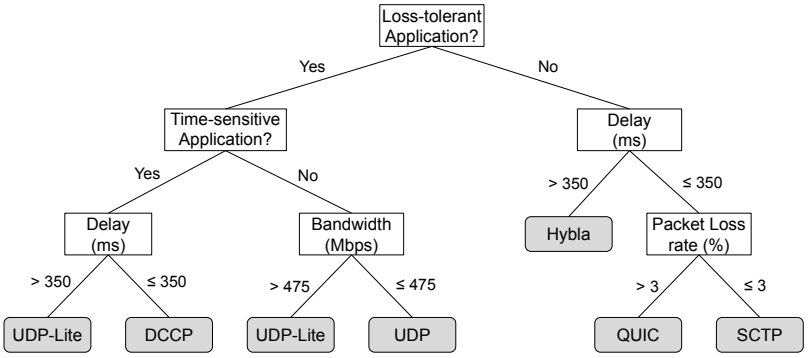


Figure 5.8: Extended decision tree to select the most appropriate protocol. Leaf nodes (gray box) represent classes, whereas internal nodes (white box) represent the attributes.

based the selection rules of the most appropriate protocol to replace TCP. These decision trees feed and represent VTL's knowledge base that dictates the selection rules based on the above two information attributes.

Dataset. For the models' training, we generated a labeled *dataset* of more than a hundred cases. The labels or classes are the L4 protocols and the attributes, as stated above, are the application requirements and the network conditions. Following the traditional/classical approach, we separated the dataset into two main parts: the *training dataset* ($\simeq$ 66% of the initial dataset) and the *test dataset* ($\simeq$ 33% of the initial dataset). As its name suggests, the training dataset is the part of the dataset used to train the models. Additionally, it allows evaluating the trained model's ability to classify correctly the already seen cases. What about the unseen cases? The answer to the latter concern is the task of the test dataset. It permits us to evaluate the trained model's prediction quality, i.e., the precision at which the model can classify unseen cases. The dataset is generated from extensive evaluations of all considered Transport protocols for diverse application requirements and network conditions. Suppose two or more protocols satisfy application requirements and feet the network characteristics. In that situation, the performance criterion used to assign a label to the case is the throughput experienced during data transfer.

Fig. 5.7 and Fig. 5.8 illustrates examples of the outcomes of the training stage. In the instance of Fig. 5.7, the application is considered to be either loss-tolerant (profiles 1 and 4, Fig. 5.5) or not (profiles 2 and 3, Fig. 5.5). This assumption leads to a more simplified decision tree that, as we will see later, could provide satisfactory classification and prediction quality compared to a more extended decision tree. The complete evaluation of these models' quality and their use benefits are extensively evaluated and presented in Section 5.4.

5.4 Experiments and Evaluations

The main goals of carried experiments are to evaluate the VTL's benefits (in terms of performances) on TCP applications by using decision tree models. We also assessed the precisions and recalls of the trained decision tree models used to drive the best L4 protocol selection.

5.4.1 Testbed Setup and Methodology

The experiments have been performed under a testbed constituted by two hosts linked by one router (Fig. 5.1). As in Chapter 4, each host was equipped with Intel Core i7-7500U CPUs, 3.8GiB RAM, and Qualcomm Atheros QCA6174 NIC driver. In addition to TCP and its extension Hybla, we evaluated the following IETF protocols: UDP, UDP-Lite, SCTP, DCCP2, DCCP3, and the QUIC protocol. For each protocol, we implemented a distributed application (one server and one client). The server part can stream several kinds of files with different sizes ranging from a simple 4K file text to more than 132M video or text files. The network link parameters are still emulated thanks to `netem` tool. The network parameters used during experimentations are reported in Table 5.2. For each emulated link, the random loss rate is variable between 0 and 5%.

Satellite links emulation. Often used as backup Internet links, satellite Internet is useful for critical missions such as SAR (search and rescue) operations as well as to provide Internet access in rural areas. The main characteristic of satellite links is their long delay that can cause severe performance degradation. Based on [92], we used the following parameters to emulate a satellite link between the client and the server during experiments: RTT to 600 ms, and 4 Mbps of bandwidth.

Terrestrial Internet links emulation. To emulate a classical Internet link between the server and the client, we set the bandwidth to the arbitrary value of 100 Mbps and fix the RTT to 100 ms. To estimate the average RTT value on the classical Internet, we used the WonderNetwork [93] tool to find out the mean RTT between different locations all over the world within the Internet.

Local Network links emulation. The third emulated network profile is a local network (LAN), such as a home network. The RTT is set up to the highest value 50 ms whereas the available bandwidth is 850 Mbps.

The experiments were carried out in 2 stages: *(1)* First, we compared performances of the application data transfer under each

protocol, i.e., TCP and all other protocols (UDP, UDP-Lite, SCTP, DCCP2, DCCP3, QUIC). In this first step, the application had an API allowing it to directly access each of the protocols evaluated (SCTP API, DCCP API, etc.). This stage allowed us to assess the maximum benefits achievable by using the protocol selected as the most suitable alternative to TCP according to the target application and network contexts and to generate the dataset we used to train the decision tree models. *(2)* Secondly, we repeated the same experiments by comparing TCP with each of the protocols identified by the trained models as the best alternatives to TCP. But this time, the application accesses the service of the selected protocol indirectly thanks to VTL. The application invokes the socket API of TCP, but, thanks to the redirection mechanisms (implemented by the Hooker component of VTL), it will transparently use the services of the selected protocol as an alternative to TCP.

5.4.2 Decision Tree Models Benchmarking

We started by evaluating the precision and the recall of the trained decision tree models provided in Fig. 5.7 and Fig. 5.8. We constructed the models from exactly 146 instances/cases by using an open-source C implementation [94] of the supervised machine learning algorithm C5.0 (described previously in Section 5.2). For the rest of this section, we will call the simplified decision tree illustrated in Fig. 5.7 *model1* and the extended one shown in Fig. 5.8 *model2*.

The confusion matrices of *model1* and *model2* are shown in Table 5.3 and Table 5.4, respectively. The reported results show that *model2* achieves more precision (around 10%) than *model1* when it comes to select the appropriate protocol if the pair *{application*

Table 5.3: Confusion matrices showing quality parameters of the decision tree *model1*.

(a) training dataset

		Predicted						
		Hybla	UDP	UDPLite	SCTP	QUIC	**Precision**	**Recall**
Actual	Hybla	18	0	0	0	0	100%	100%
	UDP	0	0	6	0	0	—	0%
	UDPLite	0	0	42	0	0	78%	100%
	SCTP	0	0	0	0	12	—	0%
	QUIC	0	0	6	0	24	67%	80%
	Weighted Average						**78.7%**	**93.3%**

(b) test dataset

		Predicted						
		Hybla	UDP	UDPLite	SCTP	QUIC	**Precision**	**Recall**
Actual	Hybla	6	0	0	0	0	100%	100%
	UDP	0	0	2	0	0	—	0%
	UDPLite	0	0	15	0	0	79%	100%
	SCTP	0	0	0	0	4	—	0%
	QUIC	0	0	2	0	9	69%	82%
	Weighted Average						**79.5%**	**93.8%**

(a) training dataset

		Predicted							
		Hybla	UDP	UDPLite	SCTP	DCCP	QUIC	**Precision**	**Recall**
Actual	Hybla	20	0	0	0	0	0	100%	100%
	UDP	0	12	3	0	0	0	67%	80%
	UDPLite	0	3	15	0	3	0	83%	71%
	SCTP	0	0	0	12	0	0	100%	100%
	DCCP	0	0	0	0	15	0	83%	100%
	QUIC	0	3	0	0	0	24	100%	89%
	Weighted Average							**90%**	**89%**

(b) test dataset

		Predicted							
		Hybla	UDP	UDPLite	SCTP	DCCP	QUIC	**Precision**	**Recall**
Actual	Hybla	6	0	0	0	0	0	100%	100%
	UDP	0	4	1	0	0	0	67%	80%
	UDPLite	0	1	5	0	1	0	83%	71%
	SCTP	0	0	0	4	0	0	67%	100%
	DCCP	0	0	0	0	5	0	83%	100%
	QUIC	0	1	0	2	0	6	100%	67%
	Weighted Average							**86%**	**83%**

Table 5.4: Confusion matrices showing quality parameters of the decision tree *model2*.

requirements / network context} is already encountered. The trend is reversed for the recall's values where on weight-average, model1 presents 93% recall, whereas *model2* achieves 89% recall. As stated previously, the ability to classify correctly already seen cases is not sufficient to assess a model's quality. Its prediction quality, i.e., its ability to classify accurately new and never seen instances, gives more insights. Therefore, we apply the trained models *model1* and *model2* on a *test dataset* containing around forty cases. We observed that *model1* classify almost with the same precision (79.5%) seen as well as unseen cases. The trend is slightly different for *model2*, where the achieved precision (86%) on the unseen instances is not so better as the precision of the classification of seen cases.

All in all, we note that the simplicity of a model is not necessarily a restriction to its usage. The quality achieved by a simplified model (for instance, *model1* in our work) could be good enough for its use. A model could classify correctly all seen cases but perform worst on new and unseen instances. The trained models *model1* and *model2* are able to make accurate selection of the appropriate protocol 8 times out of 10.

5.4.3 Application Performances

Absolute throughput evaluations. In a first step, we assessed all protocols' absolute performance, i.e., without VTL operations and use of trained models. The results reported in Fig. 5.9 show the throughput of the evaluated protocols. These results are those used to generate and construct the dataset used to train the decision tree models. Furthermore, they provide us insights into what significant benefits might be achieved by using on the wire another protocol instead of TCP (as presented in Section 5.1).

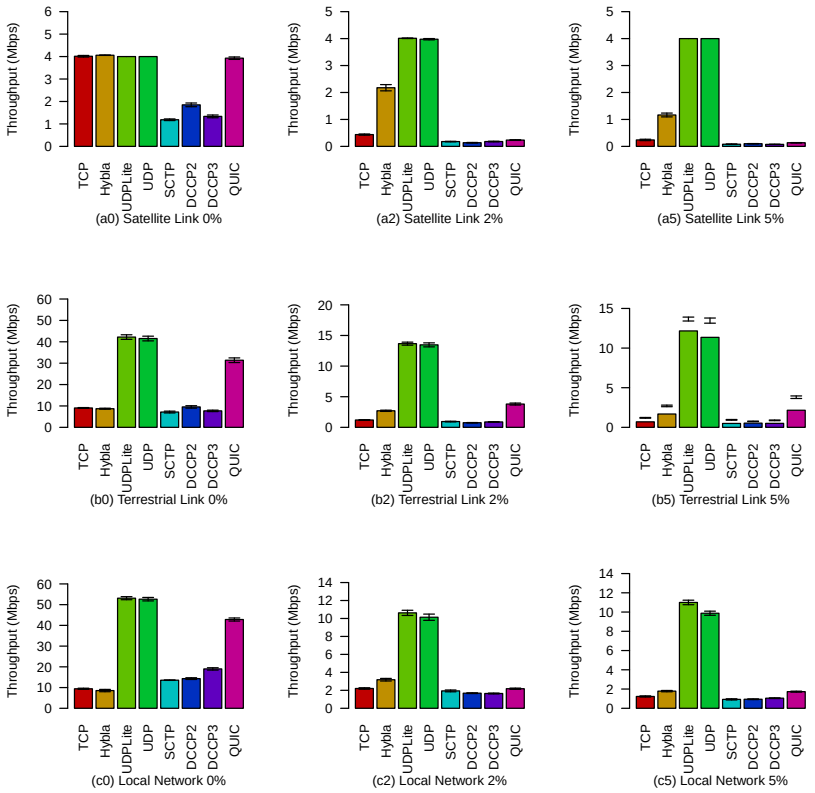


Figure 5.9: Application's absolute performance (in terms of throughput) on top of various Transport protocols.

TCP applications performance improvement. Then, we evaluated VTL impacts on the performance enhancement of TCP applications. For each considered scenario, we show only the protocols that the decision tree model selects for the considered context. For instance, when the application is loss-tolerant and the network state is {600 ms, 4 Mbps, 0%}, the selected protocol by *model1* to replace TCP is UDP-Lite. In the same network context, when the application is sensitive to data packet losses, the protocol selected by the decision tree *model1* and *model2* is Hybla. Then, Fig. 5.10 (a) compares the TCP application's performance without redirection and its performance when it is redirected to UDP-Lite or Hybla. The evaluations reported from Fig. 5.10 (b) to Fig. 5.10 (i) follow the same logic in order to alleviate the figures. The results show that VTL allows TCP applications to achieve at average ~5x better performances in most scenarios.

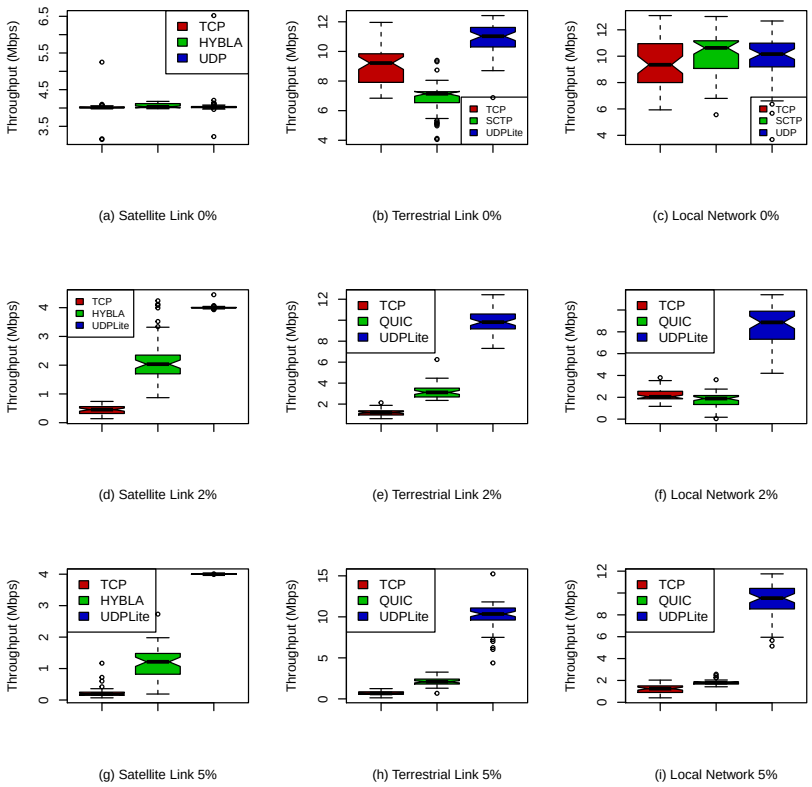


Figure 5.10: Hooked TCP Application's performance (in terms of throughput) under VTL.

5.5 Conclusion

In this chapter, we presented our approach to address the problem of the selection of the "best" Transport protocol to use rather than TCP. The choice aims to meet the application's requirements and take into account the network conditions. To perceive this information (i.e., applications needs and network state), our approach relies on the *dynamic* identification of the TCP applications' requirements and the underlying network characteristics. We used a set of machine learning models, namely decision trees that we trained to feed the knowledge base of VTL.

We evaluated our proposed solution within the context of experimental measurements based on the use of a distributed variable application profiles in an emulated environment of *(i)* satellite networks, *(ii)* long-distance terrestrial networks, and finally *(iii)* local networks. Apart from the proposed approach assessments, this measurements' campaign serves us to generate the dataset used to train priorly the decision tree models. Based on extensive evaluations of the quality parameters of the trained, we showed that on average 8 times out of 10, VTL correctly selects the best alternative to TCP. The immediate consequence of this is the improvements of the hooked TCP applications' performance under VTL. We ended by evaluating these performance gains in the different considered contexts.

Conclusion 6

Attendee. Yet another Transport protocol?
Me. No, VTL is a Transport layer architecture that envisions achieving effectively *dynamic deployment* of any Transport protocol and stimulating its *adoption* by new aware-applications and existing legacy applications.

This is a short conversation with an attendee at one of my first presentations at some academic convention. This question often appeared when I pitched the work presented in this thesis manuscript. Indeed, for several decades, the Internet's Transport layer has been the center of many research contributions leading to a *plethora* of new Transport protocols such as DCCP or SCTP, to name a few. These research contributions generally share the objective of fulfilling the QoS requirements of new applications and/or taking into the underlying networks' characteristics. The outcome is a theoretical layered architecture of the Internet illustrated in Fig. 6.1 (a). The Internet's designers expected a Transport layer with plenty of L4 protocols, each protocol serving a particular purpose in some specific contexts. There is no *one-fits-all* solution; that is to say, there is no single L4 protocol that meets *any* application requirements on *any* network conditions. Nevertheless, it is clear that TCP remains the most widely deployed and adopted Transport protocol on top of which almost 90% of the Internet's applications run. The result is the hourglass model of Fig. 6.1 (b) that perfectly illustrated the Internet's architecture in practice.

Any L4 protocol other than TCP lacks a wide deployment and/or encounters limited use on the Internet. This phenomenon is known as the *ossification* or *sclerosis* of the Internet's Transport layer that hampers the introduction of new protocol solutions at this layer of the Internet. Throughout this thesis, the contributions we presented aim to tackle this ossification of the Internet's Transport layer at the end-system. We explored and proposed technical and conceptual approaches that pave the way for the effective use of any existing Transport protocol and the dynamic deployment in the end-systems of a new one so far as the protocol meets the application's requirements and the network's characteristics. We summarize this thesis's main contributions in Section 6.1 and open up the way for further research directions in Section 6.2.

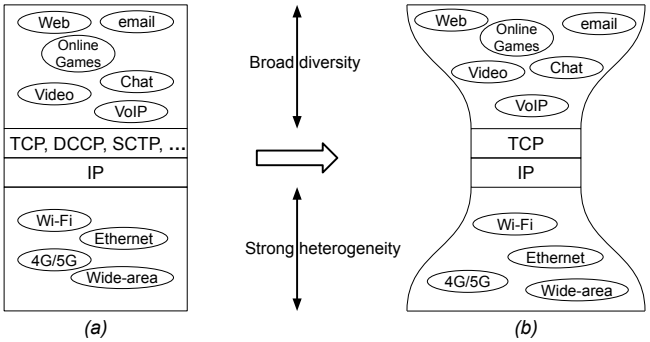


Figure 6.1: The expected ideal Internet layered architecture (a) vs. the hourglass Internet architecture in practice (b).

6.1 Contributions Dissemination in Chapters

In summary, the contributions of this thesis are disseminated in the manuscript's chapters as follows. In ***Chapter 2***, we detailed the limitations of the current Transport layer protocols and architectures to address the ossification issues. The insights from these limitations' analysis defined the guidelines for the design of a new architecture of the Internet's Transport layer. For recall, the fundamentals of these guidelines are the following. In order to limit the OS implication in the deployment of a new protocol, the new Transport layer system should provide a ***safe and isolated runtime environment*** where the protocol functions are dynamically deployed without the need to recompile the OS. To prevent the limited adoption on the Internet by application developers of the new Transport layer, the system should provide a ***protocol-agnostic interface*** to applications in the way that the latter are removed from the complexity of choosing the L4 protocol to use. Further, we should allow the legacy applications to *transparently* use the new Transport layer, that is to say, the existing applications do not have to be rewritten. Following these guidelines, we designed and implemented a novel Transport layer called VTL for Virtual Transport Layer.

In ***Chapter 3***, 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.

In ***Chapter 4***, we presented the design of the *Hooker*, a VTL component that enables the replacement of TCP with another Transport protocol during data transfer. We performed TCP's replacement *transparently* for legacy applications, i.e., there is no need to modify these applications. The goal is to comply with the *transparency* requirement learned from the guidelines provided in Chapter 2. Fulfilling this requirement is a key factor in promoting the use of VTL and its associated Transport protocols either deployed by VTL or already existing in the end-system OS. We performed extensive evaluations to show the effectiveness of the *Hooker* solution, i.e., its ability to replace at *runtime* TCP by an alternative L4 protocol *X* without any modification of the legacy application. Finally,

we assessed *Hooker's* impact on TCP application's performance. The results showed that the most appropriate alternative protocol to TCP varies depending on the network conditions and the legacy application's requirements. Further, we noted that the selected alternative protocol's performance could even be worse than TCP's one if the alternative is chosen without thought.

Therefore, in ***Chapter 5***, we presented our last contribution that addresses the problem of the selection of the "best" Transport protocol to use in replacement to TCP based on the application's requirements and the network conditions. To perceive this latter information, we proposed and implemented *dynamic* identification algorithms of the TCP applications' needs and the underlying network characteristics. We used a set of machine learning models, namely decision trees that we trained to guide the best protocol selection. Finally, we carried out thorough assessments of our proposed algorithms and models. The evaluations showed that leveraging the trained models feeding its knowledge base, VTL accurately selects the most appropriate Transport protocol for diverse application profiles on different network conditions. The outcome is the improvement of the hooked TCP applications' performance.

All in all, the work we carried out during this thesis journey contributes to replace the *vicious* circle (described in Chapter 2) with a *virtuous* circle where the Transport layer *(1)* provides an isolated, efficient, and flexible environment for dynamic deployment of any protocols within the OS, and *(2)* allows application programmers, without modifying their applications, to take advantage of new Transport solution so far as the new solution meets their needs.

6.2 Potentials Future Work

The work carried out in this thesis opens up a way for potential further research efforts. We outline the most relevant of these works by structuring them into short-term works and then into mid and long term works. We consider as short-term perspectives the work that can be carried out in the direct prolongation of this thesis and classify as long-term perspectives, the work whose realization would require going beyond the scope of this thesis's initial objectives.

We situate the considered perspectives in three main interdependent research directions which are as follows:

Heterogeneity of the context. In work presented in this manuscript, we have implicitly addressed heterogeneity at the level of applications (QoS needs, legacy vs. aware, etc.) and at the level of networks (presenting different characteristics). However, in order to consider the use of VTL in a more extended context, the problem of heterogeneity must be addressed more broadly by extending it, for example, to the level of host machines whose capacities and resources might be different.

Dynamic and autonomous (re)-configuration of protocols. The ability to (re)-define and (re)-deploy protocols at the right moment (e.g. when the context changes) is a fundamental challenge for VTL. In this perspective, the discovery of opportunities and constraints of the context or the choice of protocol deployment modalities are necessary functionalities beyond those currently covered by VTL. Consequently, the autonomy of the VTL, i.e. its capacity to operate its actions with the minimum of human intervention, is a relevant perspective to be explored in greater depth beyond the question of choosing the best protocol proposed in this thesis.

Scalability and VTL. Finally, the modalities of VTL deployment and instantiation must be questioned in the case an increasing number of transport sessions are to be established. Would it make more

sense to activate one instance/agent per end-system, per application, or per connection? The answer to these concerns meets, here also, the objective of using VTL in a more general context. The resistance to the scale factor of the VTL with respect to the number to a high number of connections at the host machine level, is a property to be considered in relation (in particular) to the resources of the VTL deployment nodes.

6.2.1 Short-term Perspectives

Protocol mechanisms and performance evaluation. The protocol mechanisms we have relied on (especially in Chapter 3) 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 protocol or recent congestion control mechanisms such as PCC [95] 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. It would also involve assessing the VTL within more elaborate case studies and considering other performance metrics such as per data packet transit delay, jitter, or fairness and friendliness of the Transport session flows. The deployment of a connection involving a TCP-based VoD server and a VoD client using (thanks to VTL) a protocol other than TCP, but compatible with the latter, is an illustrative case study.

eBPF technology. Although conceptually robust, eBPF technology presents at some places limitations related to the current implementation choices of some of its components, notably `SOCKMAP` (described in Chapter 4). For example, 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.

6.2.2 Mid and Long Term Perspectives

General perspectives concerning the need for heterogeneity management

In the emerging Internet, we could expect some nodes might have VTL, and others will not. Similarly, some nodes will be able to allow on-demand deployment of VTL, and others will not. The communication opportunities between "VTL nodes" and "non-VTL nodes" raises a relevant challenge that must be tackled to allow the interoperability of different protocols with the objective (for example) to optimize QoS.

Potential VTL deployment nodes will present different resources in terms of usable Transport services and protocols deployment technologies. Some technologies might allow the deployment in the user-space whereas others might permit it in the kernel-space. On this line, operating systems will present different capabilities/opportunities (e.g., eBPF in the Linux world). It is then necessary to study when, how, and under what conditions to leverage these opportunities' heterogeneity, especially in terms of deployment technologies. It is advisable, where the technologies are sufficiently mature, to initiate work similar to the one carried out in this thesis (around eBPF) in other OS

contexts, starting, for example, from work opened in [96]. The question of the choice among these protocol deployment opportunities will thus have to be addressed.

Finally, the generalization of *cloud computing* capabilities will gradually make it possible to deploy *virtualization containers* (*virtual machine* or *container*) on end nodes. One perspective is, therefore, to define how to take advantage of this capacity to deploy VTL where it is not present by leveraging *softwarization* and network programming technologies, namely SDN and NFV (introduced in Chapter 1). As a result, the VTL architecture needs to be revisited to enrich it with the necessary functionalities.

General perspectives concerning the need for autonomous (re-)configuration management

Architecture of the VTL. The architecture of the VTL is currently designed to allow the dynamic deployment of protocol components in the kernel (via eBPF on Linux OS) based on the acquisition of application requirements and network characteristics. At the same time, the reconfiguration dimension is only partially addressed.

Enriching the VTL is a fundamental perspective with the aim to: 1/ discovering the context and its opportunities (e.g. in terms of deployment capacity in the kernel or user-space, available Transport services, the presence or absence of middleboxes on the data path, etc.), and 2/ choosing, as a consequence, the configuration methods for protocol solutions, and the solutions themselves. A work extending the design and implementation of the VTL components might be carried out based on the available technological evolutions and the targeted degree of autonomy for the VTL. Concerning this last point, approaches based on the autonomic computing model have already been pushed forward. It would be a matter of re-studying them in the new context considered.

Offline learning vs. Online learning. During our work, the machine learning models used to select the most appropriate Transport protocol have been trained *offline* beforehand of the deployment of the VTL system. A future direction could be to enhance this approach with *online* learning. That is to say, VTL should be able to learn and update alone the initially trained models. This will permit to limit the risk of inaccurate models when the network environments radically changed or integrated new characteristics not considered in the initial training.

General perspectives concerning the need for VTL scalability management

The VTL was designed without any preconceived expectations in terms of scalability concerning the number of connections likely to benefit simultaneously from VTL services. However, it is clear that a VTL inability to handle numerous connections "at the sufficient pace" would be a potentially significant obstacle to its deployment. Without answering the question, the subject refers to ways of considering the deployment/instantiation of VTL at the node level: per session, per application, or per end-system. Exploring these possibilities through the prism of their consequences on scalability is a perspective of our work.

Author's Scientific Production

International Conferences and Workshops

• *El-Fadel Bonfoh*, D. C. Tape, C. Chassot, S. Medjiah, "Transparent and Dynamic Deployment of Lightweight Transport Protocols", The 2019 IEEE Global Communications Conference (GLOBECOM), December 2019, Waikoloa, Hawaii, USA.
• *El-Fadel Bonfoh*, S. Medjiah, C. Chassot, Jose Aguilar, "Towards the Virtualization of Transport-level Functions and Protocols", 7th IEEE International Conference on Smart Communications in Network Technologies (Saconet), October 2018, El Oued, Algeria.

National Conferences

• *El-Fadel Bonfoh*, D. C. Tape, C. Chassot, S. Medjiah, "Déploiement et Utilisation Transparente de mécanismes protocolaires légers", CORES 2020 – 5ème Rencontres Francophones sur la Conception de Protocoles, l'Évaluation de Performance et l'Expérimentation des Réseaux de Communication, Sep 2020, Lyon, France.

Miscellaneous Publications

• *El-Fadel Bonfoh*, S. Medjiah, C. Chassot, "A Parsimonious Monitoring Approach for Link Bandwidth Estimation within SDN-based Networks", Netsoft Workshop on Approaches, Analyses, and Performance Issues in Virtualized Environments and Software Defined Networking (IEEE Netsoft PVE-SDN), June 2018, Montreal, Canada.
• C. A. Ouedraogo, *El-Fadel Bonfoh*, S. Medjiah, C. Chassot, S. Yangui, "A Prototype for Dynamic Provisioning of QoS-oriented Virtualized Network Functions in the Internet of Things", 4th IEEE International Conference on Network Softwarization (NetSoft). June 2018. Montreal, Canada.

Technical Reports / Preprints

• *El-Fadel Bonfoh*, S. Medjiah, D. C. Tape, C. Chassot, "VTL: Timely Deployment and Seamless Adoption of Network Protocols". URL: https://bit.ly/3ndzQce.

Implementations

• VTL: Virtual Transport Layer
URL_1: https://github.com/elfadel/vtl
URL_2: https://redmine.laas.fr/projects/vtl

Dans certains cas, comprendre,
c'est comprendre qu'il ne faut pas chercher à comprendre
et qu'il faut agir !

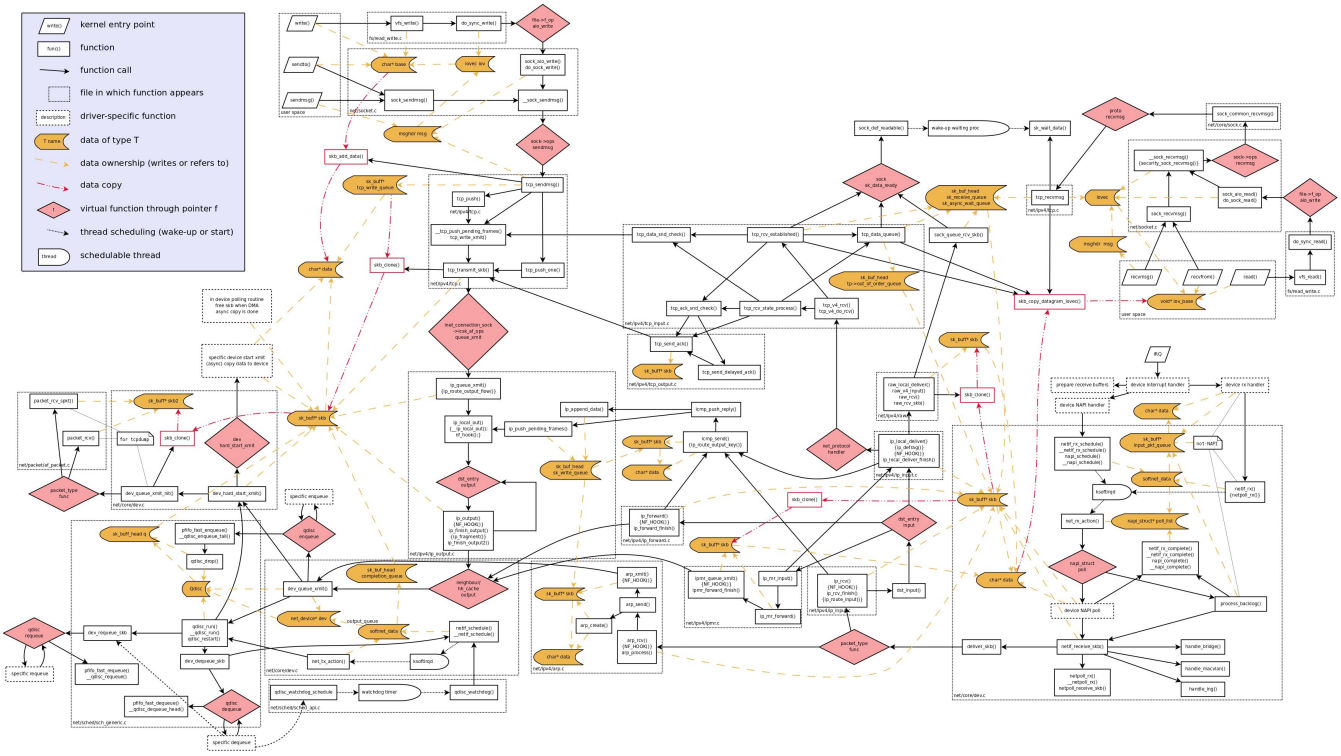
Christian MOREL

Appendix

A

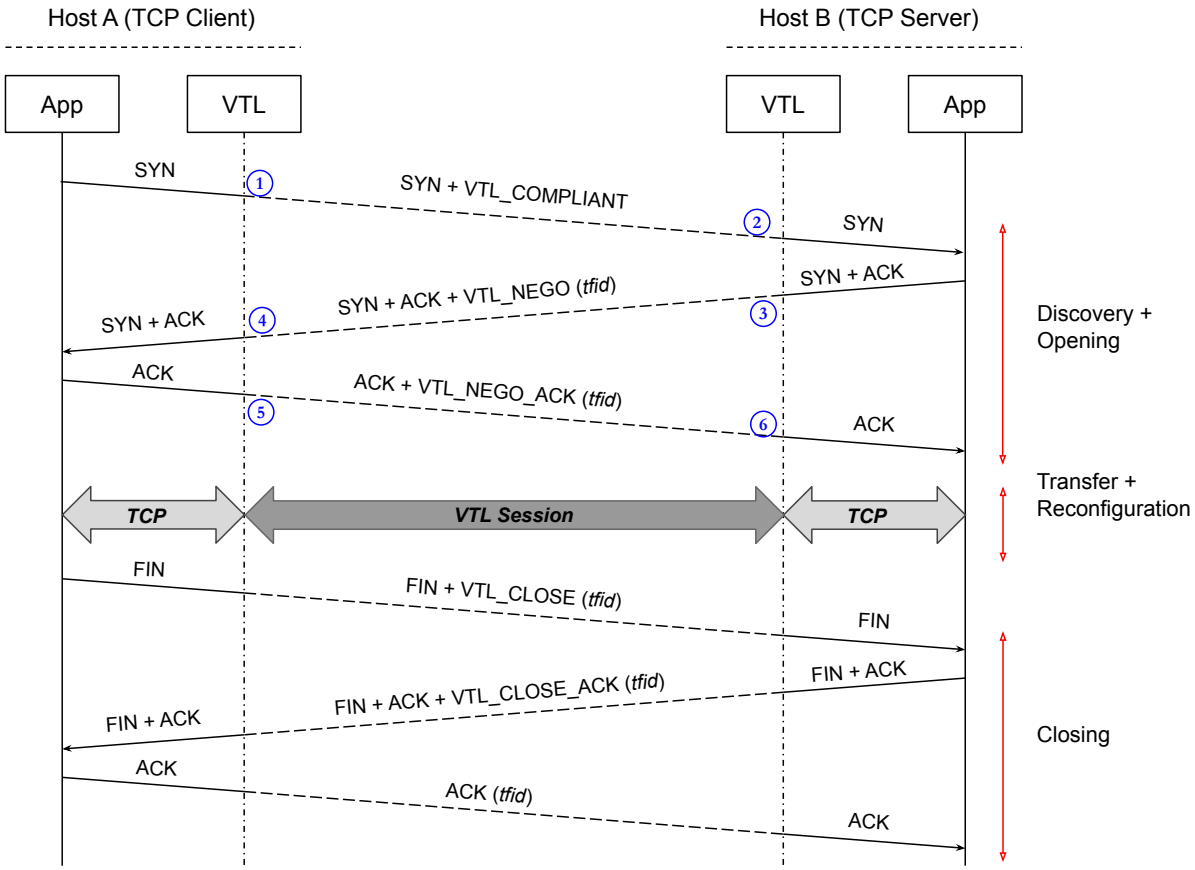
TCP Execution Path

The Figure below is derived from https://wiki.linuxfoundation.org/. It illustrates any TCP packets' complete pathway throughout the operating system (OS) kernel. Originally based on the old kernel 2.6.20, this path is still fundamentally the same on newer versions of the operating system (OS).



B

Transport Session Summary



The above figure provides an overview of a Transport session under VTL. ① When the TCP client requests a connection, its SYN packet is intercepted VTL and, thanks to a `SOCK_OPS` bpf program attached to cgroupv2, it adds to the SYN packet a VTL_COMPLIANT option to advertise to the server that the client is VTL compliant as well as to discover the server property. If the server is also VTL compliant, it should reply with a VTL_NEGO option and the transfer might continue under VTL; otherwise, the connection should fallback systematically to TCP.

② When the VTL at the server-side receives a SYN packet, it parses it thanks to an `XDP` bpf program attached to the network interface driver (NIC). If it finds a VTL_COMPLIANT option, it triggers the application profiling and gets the network state. Based on the trained decision tree models, VTL at the server-side should select the appropriate protocol to replace TCP. ③ The selected protocol, identified by *tfid* (the IP_PROTO number in fact), should be added as an option to the SYN/ACK packet. A `SOCK_OPS` program adds this option to signal to the VTL at the client-side the L4 protocol

to use for the data transfer. Here, if the application profiling fails, TCP is kept as the default protocol. One might choose another protocol as the default one.

④ Finally, the SYN/ACK packet containing VTL_NEGO and *tfid* options is intercepted by an `XDP` program at the client-side. VTL configures the requested L4 protocol to replace TCP at the client-side. Then, the ⑤ ACK of the SYN/ACK is modified by a `SOCK_OPS` program. The modification consists of adding a VTL_NEGO_ACK option to indicate to the VTL at the server-side the connection opening's success. When an `XDP` program at the server-side intercepts an ACK packet containing a VTL_NEGO_ACK option, ⑥ it removes this option to keep transparency vis-à-vis the legacy application.

Bibliography

[1] Sibylle Schaller and Dave Hood. 'Software defined networking architecture standardization'. In: *Computer standards & interfaces* 54 (2017), pp. 197–202 (cited on page 1).

[2] Mehmet Ersue. 'ETSI NFV management and orchestration-An overview'. In: *Presentation at the IETF* 88 (2013) (cited on page 1).

[3] Matt Fleming. 'A thorough introduction to eBPF'. In: *Linux Weekly News* (2017) (cited on pages 1, 22, 31).

[4] David Murray et al. 'An analysis of changing enterprise network traffic characteristics'. In: *2017 23rd Asia-Pacific Conference on Communications (APCC)*. IEEE. 2017, pp. 1–6 (cited on page 2).

[5] Laura Chappell. *Wireshark network analysis*. Podbooks. com, Llc, 2012 (cited on pages 3, 31, 66).

[6] Tommy Pauly et al. 'An architecture for transport services'. In: *Internet-Draft draft-ietf-taps-arch-00, IETF* (2018) (cited on pages 4, 17, 22, 53).

[7] David D Clark. *Modularity and efficiency in protocol implementation*. Tech. rep. RFC 817, July, 1982 (cited on pages 8, 16).

[8] David D Clark. *Designing an Internet*. MIT Press, 2018 (cited on pages 8, 22).

[9] Douglas Freimuth et al. 'Server Network Scalability and TCP Offload.' In: *USENIX Annual Technical Conference, General Track*. 2005, pp. 209–222 (cited on page 8).

[10] Martina Zitterbart. 'A multiprocessor architecture for high speed network interconnections'. In: *IEEE INFOCOM'89, Proceedings of the Eighth Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE Computer Society. 1989, pp. 212–213 (cited on page 8).

[11] Hemant Kanakia and David Cheriton. 'The VMP network adapter board (NAB): High-performance network communication for multiprocessors'. In: *Symposium proceedings on Communications architectures and protocols*. 1988, pp. 175–187 (cited on page 8).

[12] Jeffrey C Mogul. 'TCP Offload Is a Dumb Idea Whose Time Has Come.' In: *HotOS*. 2003, pp. 25–30 (cited on page 8).

[13] Michio Honda et al. 'Rekindling network protocol innovation with user-level stacks'. In: *ACM SIGCOMM Computer Communication Review* 44.2 (2014), pp. 52–58 (cited on pages 8, 16, 21).

[14] Chandramohan A Thekkath et al. 'Implementing network protocols at user level'. In: *IEEE/ACM Transactions on Networking* 1.5 (1993), pp. 554–565 (cited on page 8).

[15] Kenichi Yasukata et al. 'StackMap: Low-Latency Networking with the {OS} Stack and Dedicated NICs'. In: *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*. 2016, pp. 43–56 (cited on page 8).

[16] EunYoung Jeong et al. 'mtcp: a highly scalable user-level {TCP} stack for multicore systems'. In: *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 2014, pp. 489–502 (cited on page 8).

[17] Linux Foundation. *Myth-busting DPDK in 2020. Revealed: the past, present, and future of the most popular data plane development kit in the world*. Tech. rep. White Paper, 2020 (cited on page 9).

[18] Luigi Rizzo. 'Netmap: a novel framework for fast packet I/O'. In: *21st USENIX Security Symposium (USENIX Security 12)*. 2012, pp. 101–112 (cited on page 9).

[19] David Coffield and Doug Shepherd. 'Tutorial guide to Unix sockets for network communications'. In: *Computer Communications* 10.1 (1987), pp. 21–29 (cited on page 9).

[20] Jerome H Saltzer, David P Reed, and David D Clark. 'End-to-end arguments in system design'. In: *ACM Transactions on Computer Systems (TOCS)* 2.4 (1984), pp. 277–288 (cited on page 10).

[21] Michele Polese et al. 'A survey on recent advances in transport layer protocols'. In: *IEEE Communications Surveys & Tutorials* 21.4 (2019), pp. 3584–3608 (cited on pages 13, 21).

[22] Ernesto Exposito. *Advanced Transport Protocols: Designing the Next Generation*. John Wiley & Sons, 2013 (cited on pages 13, 21).

[23] Yujuan Jiang, Bram Adams, and Daniel M German. 'Will my patch make it? and how fast? case study on the linux kernel'. In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2013, pp. 101–110 (cited on pages 14, 33).

[24] Alan Ford et al. *TCP extensions for multipath operation with multiple addresses*. Tech. rep. RFC 6824, 2013 (cited on pages 14, 19).

[25] Nicolas Van Wambeke et al. 'ATP: A Microprotocol Approach to Autonomic Communication'. In: *IEEE Transactions on Computers* 62.11 (2012), pp. 2131–2140 (cited on pages 14, 26, 48).

[26] Jim Roskind. 'QUIC: Multiplexed stream transport over UDP'. In: *Google working design document* (2013) (cited on pages 14, 53).

[27] Adam Langley et al. 'The quic transport protocol: Design and internet-scale deployment'. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 2017, pp. 183–196 (cited on pages 14, 53).

[28] Ernesto Exposito, Patrick Senac, and Michel Diaz. 'FPTP: the XQoS aware and fully programmable transport protocol'. In: *The 11th IEEE International Conference on Networks, 2003. ICON2003.* IEEE. 2003, pp. 249–254 (cited on page 14).

[29] J Ernesto and G Exposito. 'Design and Implementation of a QoS Oriented Transport Protocol for Multimedia Applications'. PhD thesis. PhD dissertation, Institut Nat'l Polytechnique de Toulouse, Networks and . . ., 2003 (cited on page 14).

[30] Sally Floyd and Eddie Kohler. *Profile for datagram congestion control protocol (DCCP) congestion control ID 2: TCP-like congestion control*. Tech. rep. RFC 4341, March, 2006 (cited on page 15).

[31] Mark Handley et al. *TCP friendly rate control (TFRC): Protocol specification*. Tech. rep. RFC, 2003 (cited on page 15).

[32] *QUIC Implementations*. `https://bit.ly/3nTFiBf`. accessed 2020-11-24 (cited on page 15).

[33] Richard W Watson and Sandy A Mamrak. 'Gaining efficiency in transport services by appropriate design and implementation choices'. In: *ACM Transactions on Computer Systems (TOCS)* 5.2 (1987), pp. 97–120 (cited on page 16).

[34] Sebastian Gallenmüller et al. 'Comparison of frameworks for high-performance packet IO'. In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2015, pp. 29–38 (cited on page 16).

[35] Giorgos Papastergiou et al. 'De-ossifying the internet transport layer: A survey and future perspectives'. In: *IEEE Communications Surveys & Tutorials* 19.1 (2016), pp. 619–639 (cited on page 16).

[36] Mohamed Oulmahdi. 'Architecture Autonome et Extensible pour une Couche de Transport Évolutive. Application aux Communications Aéronautique par Satellites'. PhD thesis. INSA de Toulouse, 2017 (cited on page 17).

[37] Naeem Khademi et al. 'NEAT: a platform-and protocol-independent internet transport API'. In: *IEEE Communications Magazine* 55.6 (2017), pp. 46–54 (cited on pages 17, 18).

[38] Costin Raiciu et al. 'How hard can it be? designing and implementing a deployable multipath {TCP}'. In: *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 2012, pp. 399–412 (cited on page 19).

[39] D Borman. *TCP options and maximum segment size (MSS)*. Tech. rep. RFC 6691, July, 2012 (cited on page 19).

[40] Yuchung Cheng et al. 'Rfc 7413-tcp fast open'. In: (2014) (cited on page 19).

[41] Bryan Ford, Pyda Srisuresh, and Dan Kegel. 'Peer-to-Peer Communication Across Network Address Translators.' In: *USENIX Annual Technical Conference, General Track*. 2005, pp. 179–192 (cited on pages 19, 20).

[42] Rohan Mahy, Philip Matthews, and Jonathan Rosenberg. *Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun)*. Tech. rep. RFC 5766, 2010 (cited on page 20).

[43] Dan Wing et al. 'Session traversal utilities for NAT (STUN)'. In: *RFC5389, October* (2008) (cited on page 20).

[44] Gary T Wong, Matti A Hiltunen, and Richard D Schlichting. 'A configurable and extensible transport protocol'. In: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*. Vol. 1. IEEE. 2001, pp. 319–328 (cited on page 26).

[45] Linux Page. *Raw sockets*. `https://bit.ly/3l68sLR`. accessed 2020-11-25 (cited on page 31).

[46] Magnus Karlsson and Björn Töpel. 'The path to DPDK speeds for AF XDP'. In: *Linux Plumbers Conference*. 2018 (cited on pages 31, 39).

[47] Steven McCanne and Van Jacobson. 'The BSD Packet Filter: A New Architecture for User-level Packet Capture.' In: *USENIX winter*. Vol. 46. 1993 (cited on page 31).

[48] Felix Fuentes and Dulal C Kar. 'Ethereal vs. Tcpdump: a comparative study on packet sniffing tools for educational purpose'. In: *Journal of Computing Sciences in Colleges* 20.4 (2005), pp. 169–176 (cited on page 31).

[49] Cilium. *BPF and XDP Reference Guide*. `https://docs.cilium.io/en/v1.5/bpf/`. accessed 2019-05-09 (cited on pages 32, 33).

[50] Toke Høiland-Jørgensen et al. 'The express data path: Fast programmable packet processing in the operating system kernel'. In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. 2018, pp. 54–66 (cited on page 33).

[51] Bert Hubert et al. 'Linux advanced routing & traffic control HOWTO'. In: *Netherlabs BV* 1 (2002) (cited on pages 33, 47).

[52] Linux Foundation. *sk_buff*. `https://bit.ly/37bqqHw`. accessed 2020-11-25 (cited on page 33).

[53] Wolfgang Mauerer. *Professional Linux kernel architecture*. John Wiley & Sons, 2010 (cited on page 34).

[54] Peter Jay Salzman, Michael Burian, and Ori Pomerantz. *The linux kernel module programming guide*. 2007 (cited on page 34).

[55] Marc Moeneclaey et al. 'Throughput optimization for a generalized stop-and-wait ARQ scheme'. In: *IEEE transactions on communications* 34.2 (1986), pp. 205–207 (cited on page 42).

[56] Stephen Hemminger et al. 'Network emulation with NetEm'. In: *Linux conf au*. 2005, pp. 18–23 (cited on page 45).

[57] Joyce Reynolds and Jon Postel. *RFC1700: Assigned Numbers*. 1994 (cited on page 46).

[58] Linux manuel page. *time - time a simple command or give resource usage*. https://bit.ly/3nStJub. accessed 2020-11-25 (cited on page 46).

[59] Alexey Kuznetsov. *IPROUTE2 utility suite howto*. 1998 (cited on page 47).

[60] Ming-Chieh Lee and Wei-ge Chen. *Video coding using adaptive coding of block parameters for coded/uncoded blocks*. US Patent 5,946,043. Aug. 1999 (cited on pages 48, 66).

[61] Parveen 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 (cited on page 50).

[62] Parveen Patel et al. 'TCP Meets Mobile Code.' In: *HotOS*. 2003, pp. 31–36 (cited on page 50).

[63] Trevor Jim et al. 'Cyclone: A Safe Dialect of C.' In: *USENIX Annual Technical Conference, General Track*. 2002, pp. 275–288 (cited on page 50).

[64] Quentin De Coninck et al. 'Pluginizing quic'. In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 59–74 (cited on page 50).

[65] E Dubois et al. 'Enhancing TCP based communications in mobile satellite scenarios: TCP PEPs issues and solutions'. In: *2010 5th advanced satellite multimedia systems conference and the 11th signal processing for space communications workshop*. IEEE. 2010, pp. 476–483 (cited on page 53).

[66] Eddie Kohler et al. 'Datagram congestion control protocol (DCCP)'. In: (2006) (cited on page 53).

[67] Randall Stewart et al. *Stream control transmission protocol*. 2007 (cited on page 53).

[68] Mohammad Alizadeh et al. 'Data center tcp (dctcp)'. In: *Proceedings of the ACM SIGCOMM 2010 conference*. 2010, pp. 63–74 (cited on page 53).

[69] Carlo Caini and Rosario Firrincieli. 'TCP Hybla: a TCP enhancement for heterogeneous networks'. In: *International journal of satellite communications and networking* 22.5 (2004), pp. 547–566 (cited on pages 53, 70).

[70] M Zitterbart. 'Parallel Protocol Implementations an Transputers-Experiences with OSI TP4, OSI CLNP, and XTP'. In: *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*. IEEE. 1992, 0_19–0_22 (cited on page 54).

[71] Stephen Lukasik. 'Why the ARPANET was built'. In: *IEEE Annals of the History of Computing* 33.3 (2010), pp. 4–21 (cited on page 54).

[72] Inge Groenbaek. 'Conversion between the TCP and ISO transport protocols as a method of achieving interoperability between data communications systems'. In: *IEEE Journal on Selected Areas in Communications* 4.2 (1986), pp. 288–296 (cited on pages 54, 55).

[73] J Griner, G Montenegro, and Z Shelby. 'RFC 3135 PILC- Performance Enhancing Proxies June 2001'. In: (2001) (cited on page 54).

[74] David Dolson, Matthew Desmond, and Jim Kuhn. *TCP proxy providing application layer modifications*. US Patent 7,277,963. Oct. 2007 (cited on page 54).

[75] Kulbir Saini. *Squid Proxy Server 3.1: beginner's guide*. Packt Publishing Ltd, 2011 (cited on page 54).

[76] T Manesh, B Brijith, TM Bhraguram, et al. 'Network forensic investigation of HTTPS protocol'. In: (2013) (cited on page 54).

[77] Michael Welzl, Florian Niederbacher, and Stein Gjessing. 'Beneficial transparent deployment of SCTP: the missing pieces'. In: *2011 IEEE Global Telecommunications Conference-GLOBECOM 2011*. IEEE. 2011, pp. 1–5 (cited on page 55).

[78] Ryan W Bickhart. *Transparent TCP-to-SCTP translation shim layer*. Tech. rep. DELAWARE UNIV NEWARK DEPT OF COMPUTER and INFORMATION SCIENCES, 2005 (cited on page 55).

[79] Gregory Detal, Christoph Paasch, and Olivier Bonaventure. 'Multipath in the middle (box)'. In: *Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization*. 2013, pp. 1–6 (cited on page 55).

[80] Stefan Klaas. *Quelques astuces avec LD_PRELOAD*. https://bit.ly/3q9xc9M. accessed 2020-11-27 (cited on page 56).

[81] Viet-Hoang Tran and Olivier Bonaventure. 'Beyond socket options: making the Linux TCP stack truly extensible'. In: *2019 IFIP Networking Conference (IFIP Networking)*. IEEE. 2019, pp. 1–9 (cited on page 59).

[82] Lawrence Brakmo. 'Tcp-bpf: Programmatically tuning tcp behavior through bpf'. In: *NetDev 2.2* (2017) (cited on page 62).

[83] Tejun Heo. 'Control Group v2. Oct. 2015. url: https://www. kernel. org/doc'. In: *Documentation/cgroup-v2. txt* () (cited on page 64).

[84] Yanlin Liu and Mark Claypool. 'Using redundancy to repair video damaged by network data loss'. In: *Multimedia Computing and Networking 2000*. Vol. 3969. International Society for Optics and Photonics. 1999, pp. 73–84 (cited on page 66).

[85] Arthur L Samuel. 'Some studies in machine learning using the game of checkers'. In: *IBM Journal of research and development* 3.3 (1959), pp. 210–229 (cited on page 71).

[86] J. Ross Quinlan. 'Induction of decision trees'. In: *Machine learning* 1.1 (1986), pp. 81–106 (cited on pages 73, 74).

[87] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014 (cited on page 73).

[88] Danilo P Mandic. 'A generalized normalized gradient descent algorithm'. In: *IEEE signal processing letters* 11.2 (2004), pp. 115–118 (cited on page 74).

[89] IT Union. 'ITU-T G. 1010: End-User Multimedia Qos Categories'. In: *G SERIES: Transmission Systems and Media, Digital System and Networks-Multimedia Quality of Service and Performance Generic and User-Related Aspects* (2001) (cited on page 76).

[90] ITUT Rec. 'X. 641 Information technology–Quality of Service'. In: *Framework* (1997) (cited on page 76).

[91] Luca Deri et al. 'ndpi: Open-source high-speed deep packet inspection'. In: *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE. 2014, pp. 617–622 (cited on page 76).

[92] *VSat System*. https://www.vsat-systems.com/. accessed 2020-10-11 (cited on pages 78, 81).

[93] *WonderNetwork*. https://bit.ly/33TWfVf. accessed 2020-10-11 (cited on page 81).

[94] J Ross Quinlan. 'Data mining tools See5 and C5. 0'. In: *http://www. rulequest. com/see5-info. html* (2004) (cited on page 82).

[95] Tong Meng et al. 'PCC proteus: Scavenger transport and beyond'. In: *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 2020, pp. 615–631 (cited on page 90).

[96] Yutaro Hayakawa. 'eBPF Implementation for FreeBSD'. In: *BSDCan 2018*. The BSD Conference, 2018 (cited on page 91).