POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master's Thesis

Design and implementation of AR/VR services at the edge infrastructure using Network Function Virtualization

Supervisor
prof. Marco Mellia

Candidate
Marco Trinelli

Tutor
Nokia Bell Labs
Massimo Gallo

April 2019

Ai miei genitori Ivan e Agata, a mio fratello Roberto, ai miei nonni Maria e Roberto, Pina e Pino. Grazie a tutta la mia famiglia, che mi è sempre vicina nonostante ci separino migliaia di kilometri di distanza.

Sommario

Introduzione e obiettivi principali

Oggi diverse applicazioni di realtà aumentata (AR) e realtà virtuale (VR) sono disponibili nel mercato tecnologico. Esempi d'uso possono essere trovati non solo in campo videoludico, come servizi di intrattenimento, ma anche in industrie manifatturiere. Utilizzando il recente Gear VR [49], l'utente viene proiettato in un'altra dimensione, ovvero la realtà virtuale, dove impersonifica e controlla un avatar in paesaggi e ambienti fantascientifici. Oppure, gli *X Glass* prodotti da X (Google)[65] sono usati nelle industrie per ottimizzare drasticamente i tempi durante i processi di produzione.

Quando si tratta di servizi AR/VR, questi necessitano sia dell'uso di elevata potenza computazionale che di bassa latenza nelle trasmissione dei dati, in modo da ottenere una fluida interazione uomo-macchina. Solitamente questi algoritmi vengono eseguiti su cluster di grandi dimensioni in infrastrutture cloud e/o fanno uso di hardware dedicato così da velocizzrae il processamento di grosse moli di dati. Tuttavia, con questa tecnica che fa uso di equipment specifico, si ottiene un prodotto finale poco flessibile e a costi elevati.

In questo lavoro presentiamo NEAR[54, 55] (Network Edge AR/VR), una soluzione che permette l'implementazione e l'utilizzo di servizi di intelligenza artificiale (AI) su dispositivi non-intelligenti (come videocamere IP, dispositivi mobili, e sistemi IoT) in maniera flessibile, economica, personalizzabile e scalabile. NEAR fa uso del *(i)* paradigma di edge-computing in modo da accelerare il processamento del servizio e garantire alta scalabilità, e *(ii)* un framework NFV the mira ad ottenere flessibilità e facilità di personalizzazione. Poiché questi dispositivi dispongono di esigue capacità di batteria e memoria nonchè bassa potenza computazionale, è necessario un terzo ente (proxy) in modo da permettere un funzionamento ottimale e performante del servizio.

Design, implementazione e funzionamento di NEAR

Una tipica architettura che sfrutta le potenzialità di NEAR è composta da un proxy server posizionato nell' infrastruttura edge, un client (smartphone, pc portatile, ...) che richiede lo stream, e un server (videocamera IP, Raspberry Pi con Pi camera, webcam, ...) che effettua lo streaming del servizio multimediale. Il proxy server, oltre a ridirigere i pacchetti tra client e server, ha il compito di eseguire il servizio AR/VR che è stato richiesto. I client hanno la possibilità di richiedere sia lo stream nativo che quello "aumentato": nel primo caso, si connettono direttamente al server; nel secondo caso, è necessario

l'utilizzo del proxy server che interviene per inserire l'informazione virtuale/aumentata nello stream. In fig. 1, un client intende guardare un video in streaming con object detection (OD) [20]. Il proxy server, avendo ricevuto la richiesta dal client, chiede il servizio "legacy" al server. A questo punto, il proxy server riceve il contenuto video come un flusso continuo in tempo reale ed esegue il servizio di realtà aumentata (OD). Poiché OD, come la maggior parte degli algoritmi di AI, processa flussi di immagini piuttosto che l'intero stream video, viene attuato un meccanismo di estrazione e re-inserimento di immagini dallo/nello stream attraverso il processo di "decoding" (per l'estrazione di immagini) e di "encoding" (per l'inserimento dell'immagine aumentata). Infine, il client riceve il video aumentato, inconsapevole dell'architettura sottostante lato proxy.

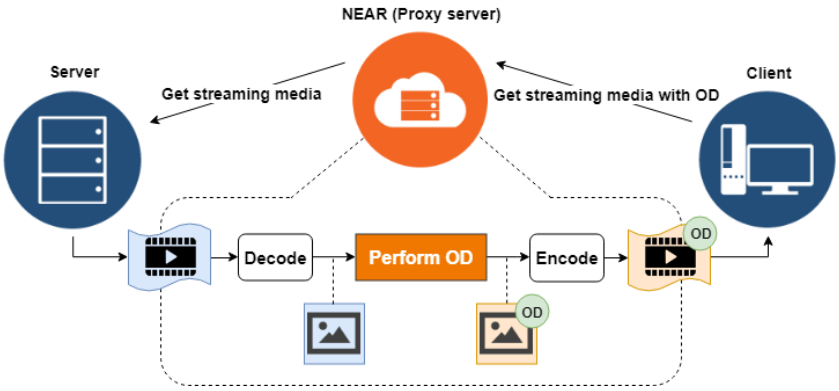


Figura 1: Design of NEAR

Server e client[1] sono due macchine con le capacità di trasmettere contenuti audiovisivi utilizzando la suite FFmpeg [21] (per il server) e ricevere il flusso multimediale tramite SOCKS [27] (per il client). Il proxy server esegue due processi, ovvero ClickNF[34, 38] che è stato scelto come framework NFV per costruire un ambiente flessibile e modulare, e la AR/VR task, incaricata dell'effettiva esecuzione dell'algoritmo di AI. La fig. 2 mostra, in verde, i nuovi elementi implementati appropriatamente interconnessi.

In un classico scenario di funzionamento, il client richiede tramite SOCKS 5 un flusso video HTTP a NEAR per mezzo di una connessione TCP con l'elemento **SOCKS 5 Proxy**. Dopodichè, NEAR apre a sua volta una connessione con il server che riceve e processa il pacchetto HTTP GET di richiesta del flusso. Quando il flusso multimediale è ricevuto da NEAR, questo viene trasmesso a **Decoder**, che fa uso delle librerie FFmpeg per effettuarne la decodifica ed estrarne un immagine. L'elemento **App Classifier** è necessario per distinguere pacchetti di controllo di SOCKS diretti verso elementi a basso livello (TCP Server) dai pacchetti di dati che devono raggiungere elementi applicativi (Decoder). Dopo la decodifica, l'immagine in uscita viene trasmessa all' **AR/VR task** (che implementa OD) per mezzo di Unix Sockets. Per ragioni di scalabilità, la AR/VR task utilizza due thread in modalità producer/consumer sincronizzati tramite costrutti di tipo

[1]Raspberry Pi con un modulo Pi Camera (server), e un pc portatile (client).

RAII. [2]. Il producer ottiene immagini da unix socket e le inserisce in una coda thread-safe. Il consumer preleva le immagini dalla coda, applica una versione personalizzata dell'Object Detection di TensorFlow e infine invia il frame aumentato a ClickNF, usando unix socket. Come ultimo passo, dato un insieme di immagini come input, l'**Encoder** ne esegue la codifica in modo da creare in video "aumentato" che è infine ricevuto dal client attraverso TCP server.

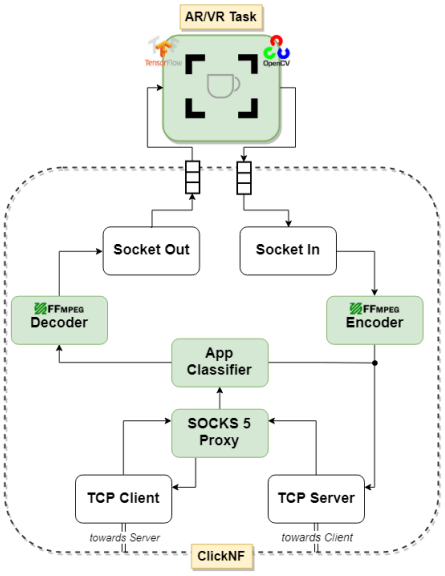


Figura 2: NEAR: proxy implementation

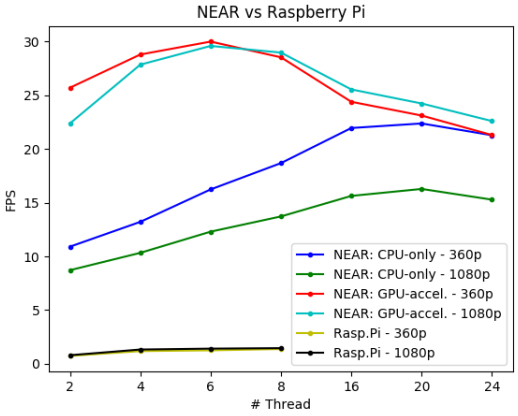


Figura 3: FPS comparison

Risultati e conclusioni

Per dimostrare i benefici ottenuti in termini di prestazioni e qualità, abbiamo paragonato (cf. fig. 3) il tasso di processamento delle immagini (FPS) a livello della AR/VR task con l'implementazione di Object Detection in un singolo dispositivo (Raspberry Pi). [3] Secondo i risultati ottenuti, l'ultimo ottiene non più di 2 FPS a differenza di NEAR che grazie anche ai benefici derivanti dall'utilizzo di una GPU di fascia alta, incrementa le prestazioni secondo un fattore moltiplicativo 10x (usando 20 threads, in modalità CPU-only) e 15x (6 threads, GPU-accelerated).

Nonostante possa ancora essere perfezionato in un futuro lavoro tramite l'implementazione del supporto al multi-stream con caching di contenuti, NEAR è una soluzione valida e pratica che permette a dispositivi non intelligenti di eseguire servizi di realtà aumentata/virtuale in modo trasparente[4], flessibile e semplice.

[2]Resource acquisition is initialization (cf. Glossario).

[3]NEAR dispone di 40 cores e 64 GB RAM, mentre il Raspberry Pi sfrutta 4 cores e 1 GB RAM.

[4]Gli endpoint necessitano di zero (server) o poche (client) modifiche in termini di HW/SW.

Summary

Introduction and main goals

Different Augmented Reality (AR) and Virtual Reality (VR) applications are nowadays available in the market. Example of use cases can be found not only in videogames as entertainment services, but also in manufacturing companies. The recent Gear VR[49] projects the user in another dimension, namely the virtual reality where the latter impersonates the avatar he is controlling, as if he was in the game. The hands-free *X Glass*, developed by X (Google)[65] is used in manufacturing to dramatically improve production, resulting in increasing the profit of the company adopting this solution.

When dealing with AR/VR services, those need both the use of huge processing power and ultra-low latency data transmission, in order to achieve convenient human-machine interactions. Usually those algorithms run on big clusters in the cloud and/or make use of special hardware to speed-up processing of big data. However, the use of such a technique exploiting specialized equipment outputs a final product with elevated costs and low flexibility.

In this work we present NEAR[54, 55] (Network Edge AR/VR), a solution that allows the implementation and usage of smart services on non-intelligent devices (IP cameras, mobile devices, IoT systems) in a flexible, cheap, customizable and scalable way. NEAR makes use of the *(i)* edge computing paradigm in order to accelerate service processing, and *(ii)* a NFV framework that addresses to achieve user-customization and flexibility. Since those devices lack battery capacity, memory space and computing capabilities, required to run Artificial Intelligence (AI) based algorithms, a third party (a proxy) is needed in order to allow an optimal functioning of the service.

Design, implementation and functioning of NEAR

A typical environment exploiting NEAR's potentialities is composed by a *proxy server*, located at the edge infrastructure, a *client* (smartphone, laptop, ...) requesting the stream, and a *server* (IP camera, Raspberry Pi with Pi camera, webcam, ...) streaming the media. The proxy server, besides relaying packets between client and server, is in charge to perform the chosen AR/VR task. Clients have the possibility to request the native or augmented/virtual stream: in the first case, they directly connect to the server; in the second scenario, a proxy server inserting augmented/virtual information is needed.

In fig. 4, a client wishes to play a streaming video content with object detection (OD)[20]. The proxy server, having received the request from a client, asks the legacy

service to the server. The proxy server receives the video content as a continuous live stream and performs the requested AR task (OD). Since OD, alongside with most of the other AI algorithms, works on images rather than the whole video, it is necessary to perform *frame extraction*, thus a decoder module (to extract a frame) and an encoder (to insert the augmented frame) are needed. At the end of the process, the client receives the augmented video, unaware of the proxy underlying architecture.

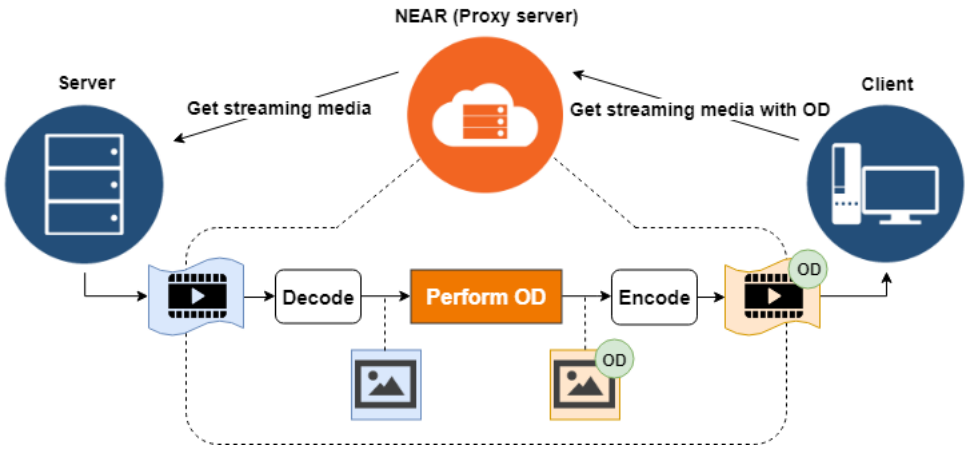


Figure 4: Design of NEAR

Alongside the design, the project has included the implementation of such three entities. Server and client are two lightweight machines (a Raspberry Pi with a Pi Camera module for the former, and a laptop for the second have been used) with the capability to stream video content using FFmpeg[21] APIs (server) and retrieve that stream via SOCKS[27] proxying (client).

The proxy server runs two processes, namely ClickNF[5], used to build a modular and flexible framework and the AR/VR task that is in charge to perform the AI algorithm. Figure 5 shows in green the new implemented elements appropriately linked together.

In a typical scenario, the client requests an HTTP video stream through NEAR using the SOCKS 5 protocol by establishing a TCP connection with **SOCKS 5 Proxy**. Afterwards, NEAR opens in turn a connection with the server, which processes the HTTP GET stream request packet and starts content streaming. When the stream is received by NEAR, it is first sent to **Decoder**, a stateful element that uses the FFmpeg APIs to decode the stream and extract a frame. The **App Classifier** element is needed to distinguish SOCKS control packets directed to lower-layer elements (i.e., TCP Server) by data packets that should reach application elements (i.e., Decoder). After decoding, the output frames are transmitted towards the **AR/VR Task** (Object Detection, OD) via Unix Sockets[6]. The AR/VR task uses, for better performances, a dual-thread producer/consumer pattern with RAII APIs[7]. The producer retrieves frames from unix sockets

[5]ClickNF[34, 38] is the chosen NFV framework.

[6]Using the Click *Socket* element.

[7]Resource acquisition is initialization (cf. Glossary).

and insert them into a thread-safe shared queue. The consumer retrieves frames from the queue, it applies a customized implementation of TensorFlow's Object Detection and it finally sends the augmented frames to ClickNF, using unix sockets. As last step, given a set of frames as input, the **Encoder** performs encoding to create the "augmented" video stream that is finally received by the client through TCP Server.

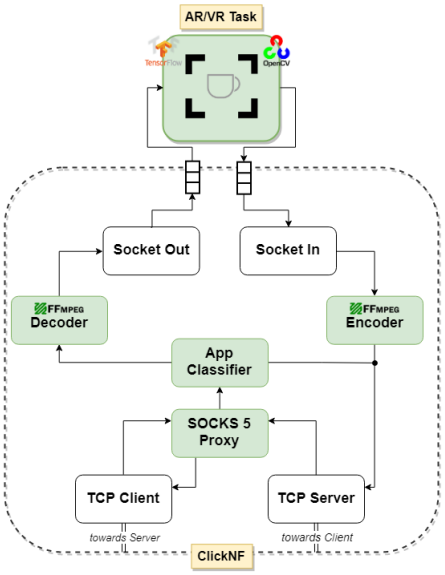


Figure 5: NEAR: proxy implementation

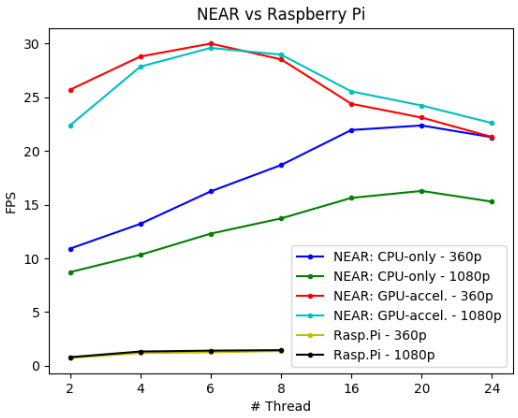


Figure 6: FPS comparison

Results and conclusions

To demonstrate the benefits obtained in terms of performance and quality we compared (cf. fig. 6) the frame processing rate (FPS) at the level of the AR/VR task with an on-single-device object detection implemented using a Raspberry Pi[8]. The latter cannot achieve more than 2 FPS, even with 4 threads. On the other hand, NEAR is able to exploit GPU-support for maximizing performances which dramatically increase by a factor of 10x (using 20 threads, in CPU-only mode) and 15x (using 6 threads, in GPU-accelerated mode).

Although it can be still enhanced, as future works, by providing multi-stream support and content caching, NEAR is a valid and practical solution that allows non-intelligent systems (IP cameras, smartphones, microphones, IoT devices, ...) to perform AR/VR services in a transparent[9], flexible and lightweight way.

[8]NEAR's proxy runs on a machine having 40 core and 64 GB RAM while the Raspberry Pi is provided with 4 cores and 1 GB RAM.

[9]Endpoints need zero (server) or quasi-zero (client) SW/HW modifications. In fact, the latter has to implement a SOCKS client only.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter introduces my work performed at Nokia Bell Labs as Intern, providing a description of the current solutions, the objectives and the main contributions of our work. It explains what are the needs that led to the initialization of the project followed by a high level structure of the thesis, that can be used to ease the reading of the work.

1.1 Context

Numerous Augmented Reality (AR) and Virtual Reality (VR) applications are nowadays available in the market. Example of use cases are very common in videogames. Just think about the (already old) Kinect [60] developed by Microsoft or the more recent Gear VR[49], the latest VR technology *made in Samsung*. Both deal with enhancing the user experience, that is projected in another dimension, namely the virtual reality. In this world a user impersonates the avatar he is controlling, as if he was in the game. As last example, the *X Glass*[1][65] aims to help everyday life. For instance, they can keep names and information about people we meet, in order to avoid eventual future embarrassments. However, today they are mostly used by employees in manufacture companies: since they are hands-free, they can dramatically improve production, resulting in increasing the profit of the company.

Anyway, in the majority of the cases, these deployments need both the use of huge processing power (we are dealing with Machine Learning algorithms based on Neural Networks) and low latency data transmission, in order to achieve real human-machine interactions. Usually those algorithms run on big clusters in the cloud. For instance, the Google Assistant [18] exploits Google's Cloud[19] platform in order to perform CPU and memory intensive jobs. Going beyond, AI-concerned devices are increasingly making use of special hardware to speed-up processing of big data. Components going towards this direction are Google's TPUs[30] and Intel's neural network processors[26].

Concerning AR/VR, we can still use special-purpose devices, built for a specific task, like the ones previously mentioned. However, first they have elevated costs and second

[1]Previously known as Google Glass. Developed by X (Google).

they are black boxes, namely, no flexibility nor customization is allowed. A second solution, presented in this work, is to use legacy devices (e.g., IP cameras, mobile devices, ...) that allow user-customization and flexibility. Moving towards this direction, the price of the device (and the service) dramatically decrease at the cost of slightly complicate the architecture used to provide the AR/VR service. In fact, when dealing with the latter, those lack battery capacity, memory space and computing capabilities, required to run AI-based algorithms. Thus a third party (a proxy) is needed in order to allow the correct functioning of the service.

In order to meet the aforementioned requirements, mainly low-latency and availability of processing power, we need to transparently delegate computation to nearby vantage points, installed by service providers, thus taking advantage of *Edge Computing*. In this way, network operators open their infrastructure to third party developers allowing them to implement services as AR/VR tasks that require, at the same time, quasi-zero-delay and CPU power.

Together with Edge Computing, Network Function Virtualization (NFV) can be used to achieve network flexibility, scalability and modularity. For instance, suppose a user wants to connect to an IP camera installed in his house and be notified when there is something suspicious. In such scenario, the network provider can dynamically allocate the resources needed to perform AI-based algorithms in its network, removing setup complexity at the front-end. In the end, the user will experience transparency (no need for a custom setup) and flexibility (given by NFV), together with quality and performance (thanks to Edge computing).

1.2 Main goals

To sum up, in this work we are going to combine the flexibility and modularity given by NFV with the performance and low-latency support powered by the edge infrastructure running Machine Learning algorithms. The objective is to allow lightweight IoT devices or legacy customized equipment to perform the same tasks, but more efficiently and conveniently, that a commercialized special-purpose AR/VR system is able to do.

It would be preferable having a transparent architecture, where the two endpoints (i.e., the media device acting as a server and the user as a client) are unaware of the underlying architecture. Both should not have any idea about the framework is being used for the content delivery. The media server, being a lightweight device (an IP camera) cannot support changes neither in software nor hardware. It will be capable to provide the basic service it has been designed for. Moreover, when connecting to the media server, the user must have the possibility to retrieve the media content as it is (*legacy service*) or to ask to the edge infrastructure to perform the chosen AR/VR task (*augmented service*).

Furthermore, since AR/VR applications usually run on mobile devices that have low computation power and risible batteries that have short lifetime, the required software needed to run those tasks must be as light as possible. In our case, edge computing based infrastructures are used to achieve this goal.

1.3 States of the arts and related works

The design and development of techniques and functionalities that allow mobile devices to exploit cutting-edge AI (Artificial Intelligence) technologies is a prolific area of study that is attracting numerous contributors. Generally, we can distinguish works that *(i)* enhance the device to accommodate AI-based functionalities, eventually using new hardware technologies[26, 30] from others that *(ii)* offload computing capabilities to third party entities (servers, proxies, GPUs, etc.).

1.3.1 Device enhancing

Belonging to the first group, CNNdroid[48] provides a GPU-accelerated library that allows to run trained Convolutional Neural Networks (CNNs) on supported mobile devices. As constraints, it currently supports Android smartphones and CNN models uniquely.

Glimpse[8] tries to bypass the design-related limitations that comes when using an on-device approach. It performs AR acceleration by filtering and sending data to the edge, to finally apply object detection[20]. Specifically, it sends compressed trigger to selected packets, and caches detections results on the mobile device for future reuse. Although increasing scalability, this technique is suitable for quasi-static videos, having consecutive frames with slightly changes.

In both the described solutions, the device is aware of the task to perform and plays the main role. Sometimes, however, a distributed architecture may be preferred, being more scalable and fault-tolerant. The second group moves towards this direction.

1.3.2 Offloading computing capabilities

Gabriel [23] can be considered part of *(ii)*. It suggests a system to support auditive-oriented tasks, targeting smart-glasses. The latter would acquire and further offload the video stream to a cloudlet, that is in charge of computing object detection and face recognition. Finally, the result is retrieved by the smart-glass that through a TTS (Text-To-Speech) component delivers the output, as an audio message, to the user.

However, despite the offloading to a cloud environment, Gabriel still suffers in terms of flexibility and scalability, being its architecture based on a set of virtual machines performing each one a single task (encoding, decoding, face recognition, etc.). This could dramatically affect scalability during parallel processing of several media streams.

Another framework, EdgeEye[37], proposes to compute an on-the-fly transformation of deep learning models to their edge computing framework. The client is constantly updated by the edge server through a real-time technology, namely WebRTC[57]. To do so, the user interacts with the application using a WebSocket API, describing the media transformation task he/she wishes to apply. In this scenario, the wide use of client-specific APIs alongside software modifications at the endpoints, make this approach non-transparent, thus rarely feasible in practice.

1.4 Contributions

As main contribution related to the context presented in section 1.1, we designed and developed **NEAR** (Network Edge AR/VR), an intelligent network architecture exploiting a NFV framework for enabling non-intelligent devices to run the latest technologies in AR and VR. NEAR has already raised interest in the research world. During its development, it has been successfully presented in public [54] resulting in having a general acceptance and interest by the audience. Moreover, our research article [55] has been positively reviewed and will be presented in the short term.

Compared to the aforementioned solutions, NEAR provides AR/VR processing acceleration through offloading computing costs to the edge gateway in a *transparent* and *flexible* way. Transparent, because it requires zero (or quasi-zero) modifications to clients requesting the service and media service providers, and flexible since it is implemented on top of a NFV (Network Function Virtualization) framework. For instance, the latter might be used to dynamically change the internal graph and thus the desired function (from simple proxying, and/or transcoding, to providing the complete AR/VR service).

Due to its 3-tier design (Server - Proxy server - Client), NEAR aims to be mainly exploited when deploying AR rather than VR services. In fact, the latter usually need a single device in order to actually achieve a virtual experience. However, we found some valid use cases for VR too. For instance, considering a scenario were Kinect is used for VR, NEAR can be exploited as a back-end framework deployed at the edge to provide flexible VR services with quasi-zero latency.

NEAR uses some already existing modules from the NFV engine, and some new components, namely network functions, that have been implemented for the purpose. Those include an **Encoder**, a **Decoder**, a **SOCKS proxy server**, a **Classifier** and an **AR task**. Those modules are a minimum set of requirements for NEAR, but a subset can be used for deploying of other applications (and projects), being each module independent from the others. For instance, a transcoding system can be set-up by using *Encoder* and *Decoder* modules only.

1.5 Structure of the thesis

This document is organized in six chapters. Chapter 1 is this introduction which is almost at the end. In chapter 2 a brief but useful description of architectures and frameworks used in the project are given for a better understating of the work. Those include a NFV framework (ClickNF), a ML platform (TensorFlow) and other supporting projects (DPDK, OpenCV, FFmpeg, ...). It follows chapter 3, that describes the design of the project. Here we present a real use case giving an idea of the high-level architecture of NEAR. Chapter 4, the most detailed chapter about the work performed during this project, presents the actual technologies used, describing NEAR's components. After that, chapter 5 shows the *testbed* used for experiments and presents the results achieved, giving pros and cons of using NEAR. Chapter 6, the sixth and last chapter, concludes the work summarizing the main aspects of NEAR, defining the achieved goals and presenting eventual future works.

Finally, at the end of the document, the reader can find a bibliography containing

some references, a glossary with the main concepts that have been used throughout the thesis and an appendix showing the algorithms of the new implemented components.

Chapter 2

Background

For a better understanding of the thesis, this chapter provides a flavor about the area the thesis project belongs to. Definitions and high-level descriptions are given, such as main concepts, architecture and framework used.

2.1 Network Function Virtualization

Network Function Virtualization (NFV) [12] is an architecture that exploits IT Virtualization to decouple network functions from proprietary hardware appliances so that they can run in software.

Without NFV, functions like NAT, Proxy, Firewalls, IDS, DNS, and Load Balancers were implemented on dedicated devices, one for each needed function. This resulted in high costs, both for equipment purchasing and maintenance, since a dedicated hardware for the specific network function was needed, low Time-To-Market, due to huge time needed to deploy those functions, and low flexibility, since it was not easy to scale up or down services running on a dedicated architecture.

NFV comes to the rescue to solve these limitations. It reduces CapEx and OpEx by removing the need to buy special purpose hardware and support facilities and eliminates wasteful over-provisioning by supporting pay-as-you-grow models. Moreover, since services are deployed via software, it accelerates Time-To-Market, and delivers agility and flexibility.

Today different NFV frameworks exist. A previous work [42] has compared several NFV frameworks [2–4, 6, 24, 28, 31, 35, 36, 38, 40, 41, 51, 53, 66] considering three main aspects: *Virtualization (V)* and orchestration of traditional services; *Performance (P)* of critical network functions for zero-delay and high throughput tasks; *Modularity (M)* of elementary network functions, that can be combined together to implement more complex tasks. Table 2.1 shows a list of NFV frameworks with their features.

NFV framework	V	P	M
ClickOS	x		x
NetVM	x	x	
SDNFV	x	x	
IN-NET	x		x
Sandstorm / Namestorm		x	
Multistack		x	x
mTCP		x	
FastClick		x	x
NBA		x	x
CoMb			x
xOMB			x
Eden			x
OpenBox			x
PDP / CliMB		x	x
ClickNF		x	x

Table 2.1: Classification of NFV frameworks

2.2 Augmented Reality and Virtual Reality

Augmented Reality (AR)[58] and Virtual Reality(VR)[64] are maybe the most fascinating discussed topics in these days. The two terms define an architecture, a device, or generally a system, that is able to enhance reality by adding information, delivering an astonishing user experience.

However, AR and VR are usually - and wrongly - used interchangeably to refer to the same concept. To be precise, VR is something more than AR. VR is able to project the user in another place through visors or goggles blocking out the room, and puts user's presence elsewhere. On the other hand, AR takes our current reality and adds something to it. It does not move us elsewhere. It "just" uses sophisticated Machine Learning (ML) algorithms to add information to our world.

To make some example, wearing a Sony PlayStation VR [62] you can be equipped with a sword and fight in an imaginary arena as you were a real swordsman. You can also play tennis with your best friend, without the need to be both physically in the same room, even if the system will project you in the same virtualized tennis court. Pokemon Go [63] is a typical example of an AR application. The "AR" is the Pokemon environment, that is built on top of a real city map (like the one showed by Google Maps). In fact, a common city map is replenished by pokemons that can be caught and gyms where players can challenge themselves. All this happens when walking in the city (the real one, where the player lives). Pokemon Go is very close to be classified as a VR application, since what is missing is the complete projection of the user to the virtual world, that can be done, with the current technologies, using a VR headset rather than a smartphone.

To conclude, even if AR can be seen as a subset of VR, they require the same advanced technology and powerful architecture that must be able to run heavy algorithms

and perform huge amount of computations on live streaming of data (e.g., video, audio).

2.3 MEC - Multi-access Edge Computing

Cloud computing [59] enables sharing rapidly provisioned services running on high performance clusters. Going beyond, Multi-access Edge Computing (MEC) [13], previously known as Mobile Edge Computing, is a computing and network architecture that offers both application developers and service providers cloud computing services at the edge of the network. MEC brings real-time, low-latency and high bandwidth access to highly-constrained applications such as Internet-of-Things(IoT), augmented reality, video analytics, data caching, connected cars, location services and more.

Using this architecture, MEC allows operators to open their Radio Access Network (RAN) to subsequently authorize third party developers to build their applications. The latter can also exploit the Virtualized Network Functions offered by the infrastructure to deliver flexibility, scalability and efficiency.

In this case, we talk about Virtualized Multi-access Edge Computing (vMEC) [43], namely a software-based solution that can be deployed on commercial off-the-shelf (COTS) machines, easily integrable with existing IT infrastructure. Using vMEC, data is first processed by virtual functions, that provide flexibility and some fault-tolerance mechanisms, since functions are software implemented. Moreover, data is consumed where it has been generated, providing the required ultra-low latency.

Nowadays MEC (and vMEC) are used when deploying infrastructure for connected cars. Those need quasi-zero latency when communicating between each other or with the network infrastructure, in order to promptly brake during dangerous circumstances (e.g., a pedestrian crossing the road).

2.4 Click

A first application that moves towards NFV is Click [32, 33]. It defines a L2-L3 modular platform for generalized and fast packet processing. In fact, functions such as L2/L3 forwarding, Checksum computation, are software implemented and they can be enabled/disabled using a configuration file.

2.4.1 Packet structure

Click's *Packet* element represents network packets that are passed from *Element* to *Element* using `Element::push()` and `Element:pull()` functions.

A Packet is composed by a *data buffer* and one or more *annotations*. The first is used to store the actual packet wire data while the second stores extra packet-related information.

Data buffer

The data buffer, shown in fig. 2.1, is represented as a flat array of bytes. The actual data can use less space than the one allocated when using the `Packet::make` function, to

create a Packet[1]. The unused space before data is called *headroom*, while trailing space is called *tailroom*. Headroom, data and tailroom are accessible through pointers (*buffer()*, *data()* and *end_data()* respectively) and sizes are given by their relative properties *headroom()*, *length()* and *tailroom()*.

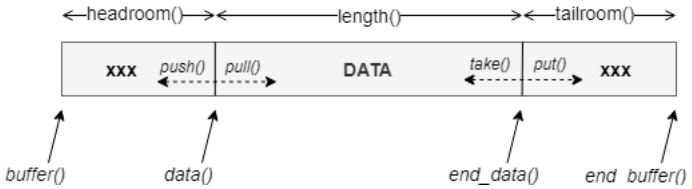


Figure 2.1: Click's Packet structure

Moreover, *push()* and *put()* methods are used to allocate extra space for headers and trailers respectively. The counterpart methods are *pull()* and *take()*.

Annotations

Annotations are metadata embedded in a packet for internal usage, such as forwarding and routing. `Packet::anno_size anno_size` specifies, for each packet, the total amount of available bytes that can be used for annotations. Elements agree to write and read portions of the annotation area in order to communicate packet-related information. Usually, to read and write packet annotations some *macros*, shared among different elements, are used.

As an example, some routing element such as *RadixIPLookup* set `Packet::dst_ip_anno` to indicate the target IP address for an ARP request. After that, this annotation is read by *ARPQuerier* to query the next hop's MAC.

2.4.2 Mode of operation

Packets in Click are exchanged between Elements through Ports whose type can be one of the following.

- **Push:** Source element creates and sends a packet to the next element. At the end, the packet arrives to the downstream destination element.
- **Pull:** The downstream destination element requests (pulls) a packet from the upstream one. If available, it returns the requested packet, otherwise null.
- **Agnostic:** The element will auto-configure that port using the same type of the other port it is connected to.

In most of the scenarios the push mode is used, since it logically follows the path made by a packet. However, in some circumstances, pull mode is needed. For instance, when

[1]Data buffer can be shared among different Packets. For a deep-copy `Packet::clone` method is used.

using a Queue element, a packet is inserted by a element using *push* but retrieved by another one with *pull*.

2.5 ClickNF

Click is a powerful modular framework that can be used as a starting point to build a NFV framework. In fact, it includes some limitations that make harder the development of a generic application.

First, it does not provides native L4 implementation. In fact, the lack of a full L2-L7 modular stack limits its scope to L2-L3 network functions. Second, it only uses non-blocking I/O forcing developers to use more complex asynchronous non-blocking I/O. Third, Click applications must rely to the OS stack, which leads to severe I/O bottlenecks due to I/O interrupts. Finally Click does not support hardware offloading and efficient timer management preventing it to scale at high-speed in particular scenarios. To overcome those limitations, a more complete NFV framework, namely *ClickNF*[34, 38][2], has been built on top of Click.

First of all, ClickNF enhances Click providing L4 modular support, including a TCP implementation that supports options, congestion control, RTT estimation, and L7 building blocks for the development of network functions. Moreover, ClickNF allows the use of *blocking I/O*, resulting in easing the development of some applications. It also offers developers standard socket, zero-copy, and socket multiplexing APIs as well as basic application layer building blocks. Finally, it improves scalability, thanks to the use of Data Plane Development Kit (DPDK) and batch processing with additional support for hardware acceleration, as well as an improved timer management system for Click.

Figure 2.2 visually shows the enhancements given by ClickNF, by comparing the latter with other traditional setups. The use of user-level libraries can exploit DMA for faster access from NIC and a modular L7 stack allows to build lightweight applications, discarding useless modules.

2.5.1 Example of a ClickNF application: SOCKS

The SOCKS protocol enables clients and servers to communicate trough a proxy independently by the upper-layer protocol used. Figure 2.3 illustrates a simplified Click configuration and the resulting graph when using an application block (*Socks4Proxy*), with some of ClickNF's building blocks (*TCPEpollServer* and *TCPEpollClient*). During initialization, Socks4Proxy configures two *bidirectional legs*, towards TCPEpollServer and TCPEpollClient respectively. After that, during data transmission, Socks4Proxy forwards packets received trough TCPEpollServer to TCPEchoClient and vice-versa.

The Click configuration file is composed by three parts. The first defines global parameters such as interfaces to use: in this case two, one to send/receive to/from client and the other for the server. The second part defines and initializes the *Elements* to be used. Input parameter, such as IP address and ports, can be given. The last section

[2]ClickNF stands for Click Network Function

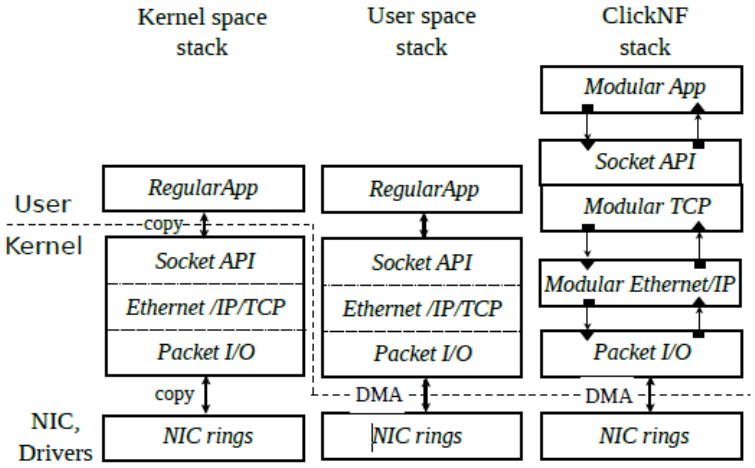


Figure 2.2: Comparison between ClickNF's and traditional stacks

specifies the actual app wiring, using *Click notation*. In this example 4 links are necessary, two between Socks4Proxy and TCPEpollServer and two between Socks4Proxy and TCPEpollClient. For instance, the first link is simply instantiated with the row `tcp_proxy[0] -> [1]tcp_epolls`, meaning that packets pushed by Socks4Proxy pushed towards port 0 are forwarded to TCPEpollServer using port 1.

Click configuration file

```
// INTERFACE DEFINITION
define($DEV0 eth1, $ADDR0 10.0.0.1, $MAC0
14:02:EC:39:48:B8)
define($DEV1 eth2, $ADDR1 10.0.0.100, $MAC1
14:02:EC:39:48:B9)

// ELEMENT DEFINITION
tcp_epolls :: TCPEpollServer($DEV1, 9000);
tcp_epollc :: TCPEpollClient($DEV0, 9000);
tcp_proxy :: Socks4Proxy(VERBOSE 1);

// APP WIRING
tcp_proxy[0] -> [1]tcp_epolls;
tcp_epolls[0] -> [0]tcp_proxy;
tcp_proxy[1] -> [1]tcp_epollc;
tcp_epollc[0] -> [1]tcp_proxy;
```

Click graph

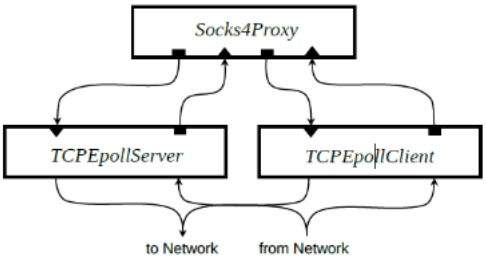


Figure 2.3: Simplified use of Socks4Proxy Element in ClickNF

2.6 DPDK and RSS

Data Plane Development Kit (DPDK) [15] is a set of libraries, running at user space, used for fast packet forwarding. Without the use of DPDK, a Network Interface Card (NIC) is normally associated to a kernel driver. In such a way, each incoming network

packet from the NIC is first copied in a buffer at kernel space[3], where header-formatting and checksum validation are performed, after that it is copied to user space where it can be used by an application. On the other hand, DPDK provides the so called "zero copy" method. The packet is directly written in user space, bypassing the kernel, beneficing both in terms of memory and time.

Generally, in most of operating systems (OS), I/O from/to peripherals is performed using *interrupts*, resulting in high overhead when acquiring the ownership of bus for data transfer. Interrupt mode is used for CPU cycles saving, since multiple I/Os are combined in one interrupt. To overcome this drawback, DPDK provides a set of APIs, under the name of Poll Mode Driver (PMD), used to speed-up NICs I/O. Using PMD, the OS will constantly spin waiting for an I/O to be available. Specifically, PMD accesses the receive and transmit descriptor queues directly without using any interrupt in order to quickly receive, process and deliver packets in the user's application. In this way, a user application does not need to wait for an interrupt to occur and can process the I/O almost instantly. Of course, this approach consumes much more CPU cycles since PMD continuously checks if I/O data is available.

Another featured provided by DPDK is Receive Side Scaling (RSS) [39], a mechanism to uniformly increase performance in a multi-CPU system. Modern NICs support multiple receive and transmit descriptor queues, usually one for each CPU. If RSS is enabled, a NIC can send the received packets to different queues in order to distribute processing among CPUs, using a filter. The filter outputs a number, between one and the number of available CPUs, that identifies the CPU the packet must be processed by. The filter is usually implemented with a hash function that takes as input the 5-tuple `(source IP, destination IP, source port, destination port, L4 protocol type)` of a packet.

2.6.1 DPDK Element in ClickNF

In the ClickNF framework, a *DPDK* Element has been designed to exploit the aforementioned advantages. The DPDK element continuously polls the NIC to fetch received packets. Before transmitting the latter, it waits for a batch of 64 packets. In such a way, Direct Memory Access (DMA) is used to directly copy batches of packets from the NIC to user level memory, resulting in amortizing the Peripheral Component Interconnect Express (PCIe) bus overhead.

The Click packet data structure has been modified in order to use a DPDK memory buffer *(mbuf)* to avoid additional memory allocation and copy operations. Each packet has a fixed size of 8 KB and consists of four sections, namely, the mbuf structure itself, packet annotations, headroom, and data. DPDK uses the mbuf for packet I/O whereas ClickNF uses annotations to store metadata (e.g., header pointers) and the headroom space to allow elements add headers.

[3]The same happens for outgoing network packets

2.7 TensorFlow

Developed at Google Brain, TensorFlow [1] is an open source mathematical library used for computation of large scale data such as Machine Learning (ML) applications (i.e., neural networks). It has been built to ease the process of acquiring data, training models, serving predictions and refining future results.

The library provides APIs for several languages such as Python, JavaScript, C/C++, Java and Go. Google suggests to use the Python API (that exploits C/C++ libraries underneath) since it is the easiest, most complete and documented one. However, in the matter of this project, C/C++ API have been used for two main reasons. First, to achieve better performance and flexibility (suggested by Google too). In fact, C/C++ TensorFlow libraries must be compiled from sources, enabling all the architectural optimizations, in order to achieve better results. Second, because ClickNF is built with that languages and a long term objective of the project could be to integrate TensorFlow in our framework. The biggest benefit TensorFlow gives is abstraction. In fact, thanks to it, developers can directly focus on the logic of the application, instead of dealing with the nitty-gritty details such as choosing which ML algorithm to use.

When building a TensorFlow application, first a model is needed, both for training and testing purposes. To run such a model, that will be exploited by the ML algorithm, a Computational Graph must be built. The following sections give a brief introduction to TensorFlow's core components.

2.7.1 Tensors

In mathematical terms, a Tensor is a N-dimensional array containing homogeneous data (i.e., only one data type is allowed). The following figure shows simple Tensors with different dimensions. A Tensor of dimension 6 is just a vector of 6 elements. A Tensor of dimensions [6,4] is a matrix of 6 rows and 4 columns and so and so forth.

The most simple values wrapped by Tensors are variables and constants. Figure 2.4 shows three examples of Tensors[4]. The first wraps a string, the second a bi-dimensional array of integers, namely a matrix, and the third a three-dimensional array of simple integers.

2.7.2 Computational Graphs

A Computational Graph (CG) is a set of *Nodes* and *Edges*. The graph cannot be cyclic, thus convergence to a node is guaranteed. An example is shown in fig. 2.5.

Each node in the graph can be a Tensor or an operation, such as addition, subtraction, multiplication, etc. Whenever an operation ends, it results in the generation of a new Tensor. The new Tensor is an update of the previous one that considers the operation performed. Operation nodes are first scheduled, and when all the dependencies are satisfied, they run through a worker process. Nodes are linked through edges to indicate

[4]Images from [1, 45].

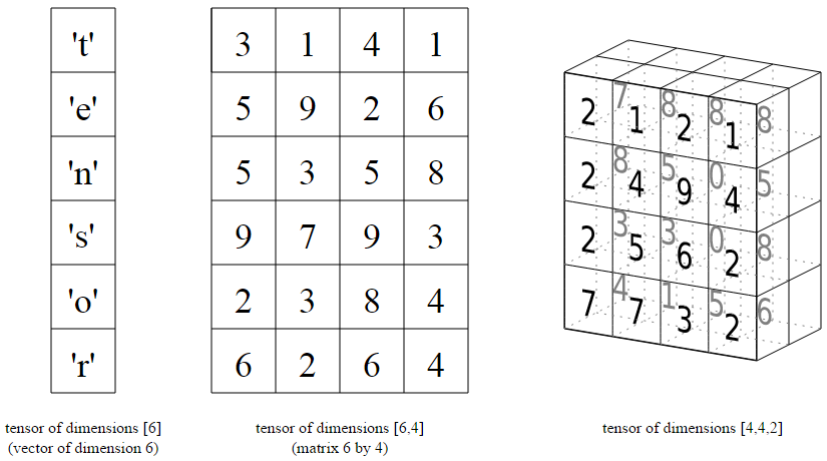


Figure 2.4: Simplified examples of a Tensor

the flow of operations. Beyond those characteristics, a feasible CG have the following features:

- Start and end nodes (like a and b in fig. 2.5) are always Tensors. In fact, an operation always needs at least one Tensor as input.
- When the CG is traversed in forward direction, the nodes encountered become a dependency for next nodes (e.g. to solve the final node e, c and d must be first computed).
- If the CG is traversed in reverse order, then the sub expressions formed can be combined to form the final expression. In fact, the final expression $e = c*d$ can be written as $e = (a+b) * (b+1)$.
- Nodes at the same level are independent each other. This means, for example, that operational nodes c and d can be computed in parallel.

Finally, a CG can be split in subgraphs. Those graphs, if in the same level, are independent and so can be run in parallel in different workers.

2.7.3 Execution of subgraphs

Thanks to the last property, parallel computing can be hugely exploited. In the previous example, operations c and d can be scheduled on two different CPUs, as shown in fig. 2.6. On the left, a TensorFlow session creates a single worker that is responsible for executing a subgraph, thus scheduling tasks on various devices. On the right, multiple workers are instantiated. They can run on the same machine or on different machines and each worker runs its own context. in the above figure, each worker runs on a different machine and schedules operations on all available devices.

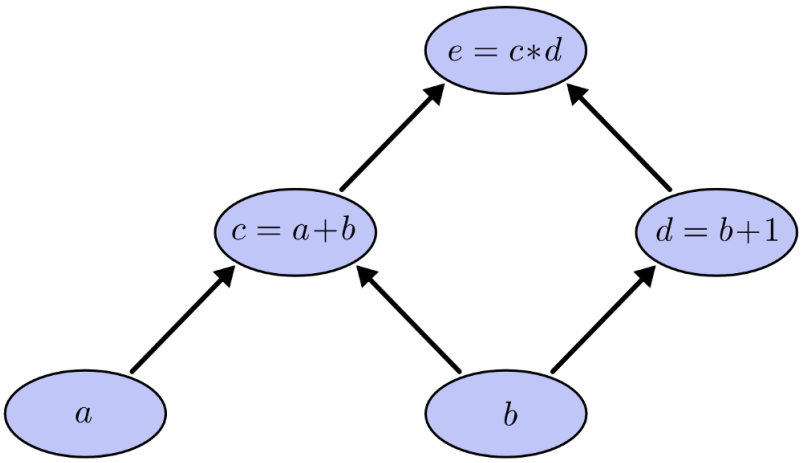


Figure 2.5: A Computational Graph

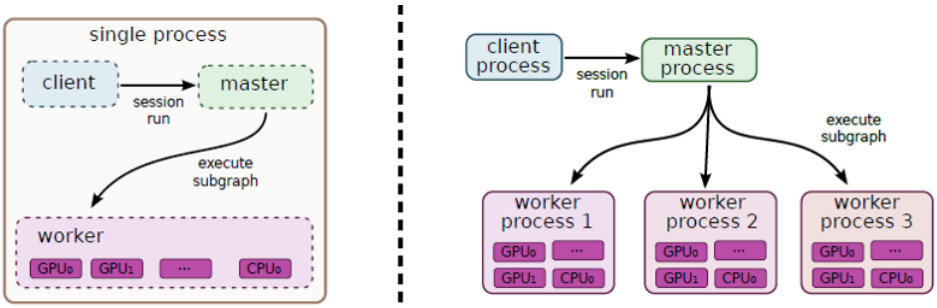


Figure 2.6: Distributed execution of subgraphs using a worker (left) or a master process that spawns multiple workers (right)

2.8 Training and Inference in Neural Networks

When talking about Artificial Intelligence (AI), we usually refer to the ability of machines to *act* based on a past experience. Obviously, machines cannot learn from nothing, they need a *trained model*, namely examples, that can be used to learn basic concepts, such as, what is a cat and what is a dog. After that, machines can *infer* (i.e., imply something new) based on previous examples. Hence, as illustrated in fig. 2.7, Deep Learning algorithms are split into two tasks: *training* and *inference*[5].

In the first, enormous datasets containing *labeled data*. Besides data, *metadata* is used as an hint to describe the related data given as input. AI tasks, due to their nature, usually run on Neural Networks (NN), a computational model inspired by the biology of humans brain. Differently from a humans' neurons that can connect to any other neuron within a certain physical distance, artificial NN have separate layers, connections, and directions of data propagation. When *training* a NN, training data is the input of the

[5]Image from [10].

first layer of the network, and individual neurons assign weights to the input, indicating how correct or incorrect it is, based on the task being performed. In the example of an image representing a cat, the other layers consider selected features (type of nose, color, ...). After, each layer passes the image to the next, that will assign other weighs considering other specific characteristics of the input data. At the end, the final layer and the final output determine the total of all those weighting. In the case the algorithm, reading those weights, informs the NN the output is wrong (e.g., the image represents a dog), a *back-propagation* mechanism takes place. The error is propagated throughout the previous layers, setting new weights for each node. This process is repeated until is found the correct set of weights (in this example, representing a picture of a cat). At the end, the training phase produces a *trained model*, that can be distributed to different machines.

During *inference*, the machine uses the previous trained model for serving predictions about new unlabeled data. in this case, no back-propagation mechanism is performed. Given the input image, weight are computed for each node at a layer, and forward propagated to next layers, until the last one, where the final weighting is produced. The set of final weights is read by the AI algorithm that, finally, is able to label the input data (in our case, a cat).

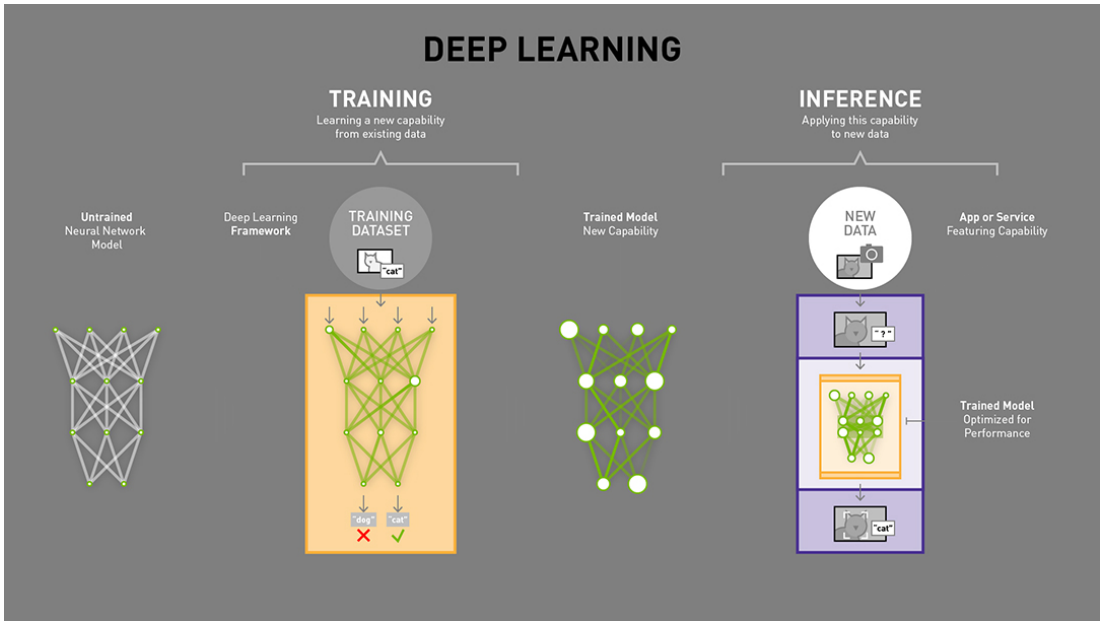


Figure 2.7: Training vs Inference in Deep Learning

2.9 OpenCV

OpenCV (Open Source Computer Vision) [11, 14] is another library related to ML area. It was developed by Intel with the intent of supporting real-time computer vision applications.

OpenCV is more than an Image Processing library. In fact, it is widely used with

common Deep Learning frameworks such as TensorFlow, Torch [9, 44] and CAFFE [29] in order to ease the development of applications and interactions between different frameworks. For example, it is exploited to easily read/write images, detect faces (cf. fig. 2.8)[61], shapes and simple objects, write dynamic text on a video, recognize text, etc. Generally, OpenCV is a good interface to support the development of AR/VR applications.

Last but not least, it is well documented and libraries are written in C++ for best performance. Moreover, bindings in other languages (such as Python, Java and MATLAB) are provided.

2.9.1 RGB and YUV frame representations

OpenCV provides support to the two most common frame representations, namely RGB and YUV format. RGB consists in the straightforward representation of the three colors red (R), green (G) and blue (B) with a given pixel size. For instance, RGB24, the most common format, allows 8 bits for each color component, that is represented by a value of 0-255.

YUV color-spaces are based on luminance. Keeping into account human perception, they allow to reduce bandwidth enabling compression artifacts to be more efficiently masked by human perception than using a RGB representation. Y' stands for the luma component (the brightness) and U and V are the chrominance (color) components. Luminance is denoted by Y and luma by Y' (' denote gamma compression) with luminance meaning physical linear-space brightness, while luma is (nonlinear) perceptual brightness.

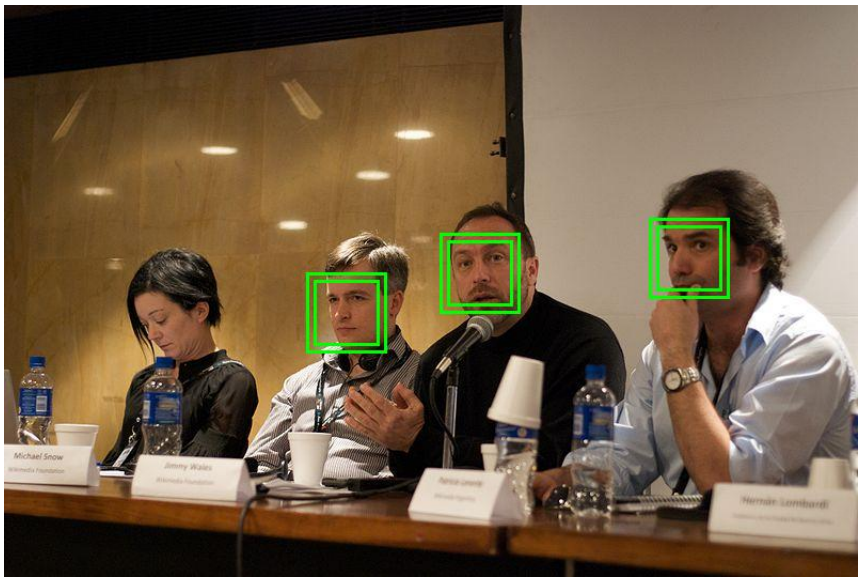


Figure 2.8: Face Detection using OpenCV

2.10 Audio and video codecs

Digital media including audio and/or video streams are packaged in a specific format using a codec. Audio and video codecs are needed to ease the storage and sharing of media files. They allow to save space, and thus download time, with a factor that ranges between three to five. Codecs can be either lossy or lossless. Lossy codecs shrink dramatically the media content, resulting in a high compression factor. Common examples are MP3, JPEG, WMA, MPEG. Loseless codecs lose zero information at the cost of a low compression factor, resulting in an encoded video size comparable to the original one. Most of the codec used for streaming are obviously lossy, since data must be sent over the network for multiple clients and a high compression factor is needed. As an example, modern codecs, such as H.265 [52], provide high compression factor without impacting data, resulting in a negligible loss to human's eyes.

2.11 FFmpeg

FFmpeg (Fast Forward mpeg) [21] is a software suite composed of libraries and tools for managing video, audio and streams. It is also the core part of other well known multimedia software such as VLC, iTunes and YouTube. FFmpeg is widely known because it provides documented APIs (in C language) for building customized applications along with different command line tools for multimedia support. The command line tools available in FFmpeg are the following.

- **ffmpeg**: It is an audio/video converter. It can also grab data from a live audio/video source (e.g. a webcam).
- **ffserver**: It is a streaming server that attaches to a ffmpeg process. It forwards ffmpeg's output via HTTP after a GET or POST request is received by a client.
- **ffplay**: It is a multimedia reader that exploits FFmpeg libraries.
- **ffprobe**: This tool is used to gather data from multimedia streams and provide information in a human-readable fashion.

2.11.1 Usage of FFmpeg suite: ffmpeg, ffserver and ffplay

Figure 2.9 illustrates a common FFmpeg architecture of an hypothetical media service provider using the three main tools of the suite: ffmpeg, ffserver and ffplay. The first runs at the input sources (a laptop with a webcam or a Raspberry Pi) and is able to provide both *offline streaming* and *live streaming*. The former delivers a media content already available in the file system, and no peripheral (camera) is needed for acquiring the input media. The latter, on the contrary, needs an on-line input camera to acquire the media source. The *ffmpeg* process is in charge of using the appropriate driver to correctly acquire the input media, perform encoding and finally create a `feed`.

An ffserver session is binded to one or more ffmpeg processes, specifically, to one or more *feed(s)*. Its main role is to make available the input sources to different users (in a LAN or throughout the Internet), eventually applying *access control*. The latter, based

on a *ffserver configuration file*, can also perform transcoding, quality adjustments and other media-related configurations. At the end of the process a `stream` (one for each input feed) is produced.

Clients can retrieve the media content by means of a media player. External media players (e.g., VLC [56]) can also be used providing that they support the streaming protocol and the codecs used by ffserver. Furthermore, to have access to the media, client must belong to the Access Control List defined by ffserver.

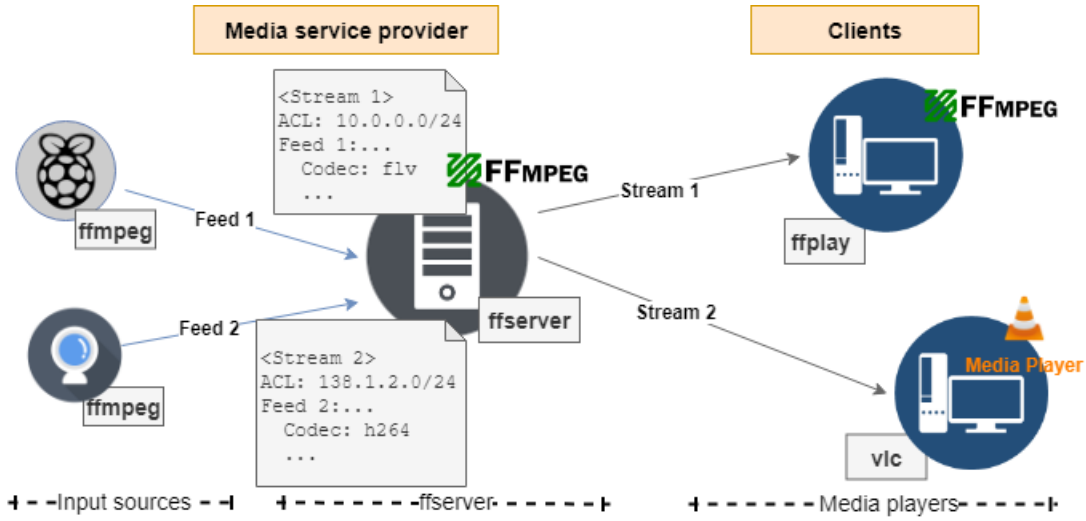


Figure 2.9: Basic usage of FFmpeg's tools

2.11.2 (De)Coding and (De)Muxing functioning

FFmpeg's libraries include APIs to manage demuxing and decoding, encoding and muxing of multimedia streams. Suppose we want to convert the format of a movie file (thus containing both audio and video streams) from *.mp4* to *.mpg*. Note that the two formats need different *containers* (MP4 and MPEG), each one supporting different A/V codecs (*MPEG-1* and *MPEG-1 Audio Layer 2* for the first, *MPEG-4 Part 10* and *AAC* for the second). The whole process can be summarized in four steps.

First, it is necessary to decouple audio and video streams from the container, and this is done by the *demuxing* function. It takes as input a container (MPEG) including audio, video and eventually subtitles, and outputs one stream for each type.

Second, the *decode* function is applied for each stream. FFmpeg's APIs use two structures: `AVPacket` to store compressed (encoded) information and `AVFrame` for decoded data. The decode function receives AVPacket(s)[6] as input and produce an AVFrame.

Third, we need to re-compress the stream, thus the *encode* function is performed. It is

[6]Depending on the used codec, the decode function must be filled up with more than one packet in order to get a frame. So, it must be called more than once.

the counterpart of the decode function: it takes as input one or more AVFrame to produce an AVPacket. Those two last steps, namely decoding and encoding, are applied for each media stream.

Finally, the output AVPacket(s) are given as input to the *muxer*. It is in charge to build a new container (MPEG-4), prepending and appending specific headers and trailers and filling it up with the new encoded streams.

2.11.3 Libraries

In order to better understand the potentialities provided by FFmpeg and ease the reading of chapter 4, it is useful to give a description of the libraries included in the suite.

- **libavcodec**: it contains FFmpeg encoders and decoders for audio, video and subtitle streams.
- **libavformat**: it provides a framework for multiplexing and demultiplexing audio, video and subtitle streams.
- **libavfilter**: it allows the video/audio to be modified or examined between the decoder and the encoder.
- **libavdecice**: it provides a generic framework for grabbing/writing from/to many common multimedia input/output devices.
- **libavutil**: it contains random number generators, data structures, hash functions, additional mathematics functions, cryptography and other multimedia related functionality.
- **libswscale**: it is used for image scaling (e.g., changing the video size), colorspace and pixel format conversion operations (e.g., convert an image from planar YUV420P to RGB24).
- **libswresample**: it performs audio resampling (i.e., changing audio rate), rematrixing (i.e., changing from stereo to mono channel) and sample format conversion operations (e.g., convert 16-bit signed samples to unsigned 8-bit).

Chapter 3

Design

Section 1.1 quickly introduced the main obstacles and the requirements that a solution to the aforementioned problem should satisfy. In this chapter, a use case followed by a detailed description of the requirements are given. After that, the designed architecture to meet the latter is presented.

3.1 A real use case

The main reason that brought us to initiate the development of this project is described in the following real scenario. Consider a terrorist, a woman with a black hat, is escaping the police in London. Two policemen are in the street to chase her, but roads are crowded, so they need help. Luckily the city has hundreds thousands cameras installed all over the roads and police can take advantage of them to get the woman. Of course, it is almost an impossible task in a crowded city like London. For humans it would be an hard task to find a woman with a black hat in the crowd, but for machines, trained over millions of samples, this task requires just few milliseconds. Thus, the two policemen would like to exploit an AI-based algorithm, like Object Detection (OD) [20] to ease the detecting of a woman with a black hat. However, neither the IP camera (a server) nor the smartphones the policemen are using (clients) can run OD, due to their low computational power and/or lack of storage.

Thankfully, network infrastructure can help in this situation. A network resources provider can instantaneously setup the resources and architecture needed for this scenario in order to transparently provide the requested service. A server located in the edge network (an edge gateway), near user's network, benefits of high computational power and thus fits the requirements to run OD. Thus, a policeman can retrieve the video stream from the IP camera by means of an OD proxy. The video will be first retrieved by the edge gateway, that will be in charge to execute OD. After that, the "augmented" video is received by the policeman that is able to identify the place where the woman is. Notice that this operation requires little modification to the existing surveillance system. Functioning of the IP camera will not be impacted at all, while clients eventually need software updates to allow to interact with the proxy.

3.2 Requirements

This section explicitly defines the requirements stemming from this use case and some possible solutions to fulfill them.

First, AI-based algorithms require high computing power and a good amount of storage requested by both the training and inferencing phases. In fact, the training phase requires to store hundreds of millions of samples with people and objects. The resulting model size, that usually directly affects accuracy, must also be stored and used at runtime, during inference.

Low battery capacity on mobile devices are another constraint. It is easy to deduct that AI algorithms consume CPU cycles and thus energy. Even considering that mobile device is equipped with the most advanced CPU in the market for mobiles, running such algorithms on it will cause such a battery drain that the device will run out of energy after few time. One solution could be to install a battery with higher capacity but this would affects size and weight and thus the convenience of the mobile.

When dealing with real-time applications, like OD, having low latency is strictly needed and maybe the most important requirement to meet. Referring to the use case, the policemen needs to detect and catch the woman now, not in an hour nor in some minutes, thus the streaming video must reach policeman's smartphone with a delay as short as possible.

Optionally, but useful to have long life and widely used, the architecture must be portable, scalable, flexible and transparent. *Portability*, because the implementation must work independently on the endpoint devices and the AI algorithm to perform. *Scalable*, because it must work independently from how many clients want to take advantage of it. *Flexibile*, in order to re-configure and thus being available during disruptions, to optimize traffic, or to efficiently satisfy users' requests. Finally, a *transparent* system would allow an easy set-up and service usage both client-side and server-side. In fact, their implementations (i.e., software and hardware equipment) need quasi-zero or no modifications.

3.3 On-device computing vs Multi-access Edge computing (MEC)

There are two possible solutions that may fulfill the aforementioned requirements. First, a centralized architecture including a special-purpose AR/VR device, with the suitable but limited computing power, battery, storage, developed for a specific task. Second, a distributed architecture, where computational power is offloaded from the device itself to a server (Edge gateway) belonging to the ISP network.

As shown in the previous use case, the first solution is not feasible, since videocameras in London are just IP cameras, and not special-purpose AR/VR devices. Even considering the feasibility of such solution (i.e., each IP camera is replaced by a AR/VR device) brings some drawbacks. First, the AR/VR device itself has a not negligible cost, much more than a simple IP camera, due to the specific design and use of recent technologies. Second, such centralized design is not scalable and there is almost no flexibility, for two

main reasons. First, performances of the device might linearly decrease with the number of clients remotely connected. Second, in case of network disruption, service availability is lost since network operators have no control on the framework that is device-embedded. In other words, both the endpoints must worry about the maintenance and setup of the service. However, using on-device computing has the benefit of achieving the lowest latency as possible, since there is no data disclosure and computation is performed locally.

The second solution, on the contrary, plans to disclose data to a third party in order to perform computation. Thus, this solution comes with an added latency overhead that was not present before. On the other hand, it is beneficial in terms of computational power and storage availability, since the AR/VR task is performed at the edge where computing and storage are not a problem.

Furthermore, thanks to modern NFV frameworks that enable programmable in-network services running at the edge, portability, scalability and flexibility are easily achieved. For instance, concerning the previous use case, consider the other policeman also wants to connect to the IP camera to help his colleague to catch the woman. Thanks to the use of NFV, it is possible to implement cross-layer optimizations (e.g., multi-client support with content caching). Thus, the video streaming is requested by the Edge gateway just once and multiplexed to the clients connected.

To conclude, using MEC the resulting endpoint devices are simple, have low cost, and are unaware of the underlying architecture. Moreover, MEC, compared to on-device computing, can lower the cost of the provided services since the latter can be shared between all the customers. In these times where AR/VR are still not so widespread and a migration from legacy to AR/VR devices is needed, using edge computing is a great solution to speed-up the deployment and lower the costs of AR/VR services.

3.4 High level design of NEAR

The design of NEAR [54, 55] (Network Edge AR/VR[1]) takes into account MEC technology running AR/VR tasks exploiting a NFV framework. The combination of MEC and NFV can be seen as the usage of a *vMEC* infrastructure, as previously described in section 2.3. Thus, we consider a *proxy server*, located at the edge infrastructure, a *client* requesting the stream, and a *server* streaming a media to the requesting client(s). The proxy server, besides relaying packets between client and server, is in charge to perform the chosen AR/VR task. Clients have the possibility to request the native or augmented stream: in the first case, they will directly connect to the server; in the second scenario, a proxy server inserting augmented information is needed. Figure 3.1 shows a high-level design.

The functioning is straightforward.

1. The client selects a proxy server and a server to connect to, specifying the stream

[1]We have found potential use cases for AR, rather than VR. But this does not mean NEAR cannot be exploited for VR tasks too. In fact, both AR and VR exploit ML techniques having same requirements.

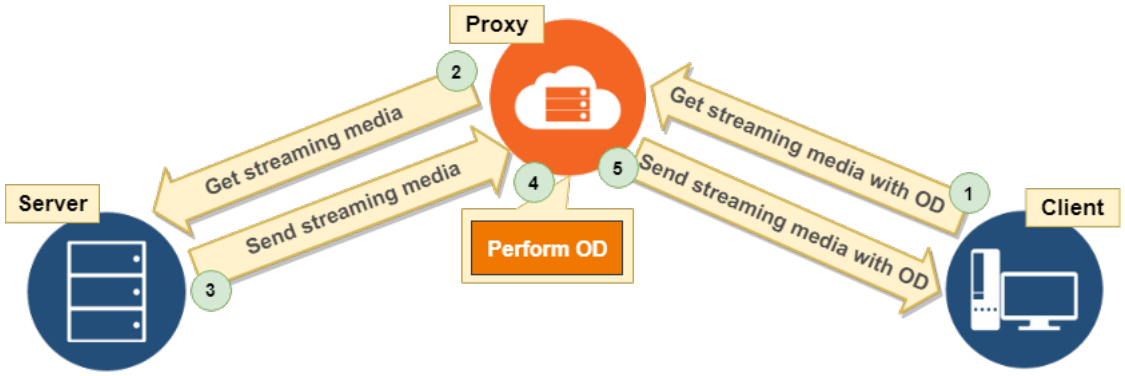


Figure 3.1: High level design of NEAR

he wants to play and the AR/VR task to perform on it, *Object Detection (OD)* for instance.

2. The proxy server sends a request to retrieve the stream to the server specified by the client.

3. The server checks if the request is valid and eventually replies with the streaming content.

4. The proxy server start receiving the streaming media. For each frame it receives, Object Detection is performed and the *augmented frame* is put into a new stream.

5. The proxy server sends the new stream, including Object Detection, to the client.

The following sections extensively describe the three elements needed, including their role and capabilities.

3.4.1 Server

The server must be able to continuously provide streaming content to clients. The latter can be either a proxy or a client directly connected. For this reason, it is strongly required that the architecture is completely transparent to the server, namely, the server is unaware about clients' capabilities and if/when/where AI-based tasks are implemented.

The server, contrary to what one might expect, does not need high computing power and elevated amount of storage, since those are offloaded to the proxy server. In order to broadcast the streaming media (a video stream with or without audio) it must be configured with a camera and a basic working network setup. A simple IP camera or a custom device including a camera, like a Raspberry PI, able to stream video content, can be used to implement the server.

3.4.2 Proxy server

The proxy server is the most complex and fully-equipped peer that is located at the edge infrastructure, namely in a location close to the client. First, it needs high performance hardware and good amount of storage in order to run AI-based algorithms. Second, it must have the server-equivalent encoding/decoding capabilities such as to perform the AR/VR task. In fact, most of AI tasks (including Object Detection) work on images rather than the whole video, so it is necessary to perform *frame extraction*. Thus, the proxy server should be able to decode the stream received by the server, apply the AR/VR task on the extracted frames and encode those to obtain the augmented stream that is finally sent to the client. Figure 3.2 shows the design of the proxy server including the subtasks to perform.

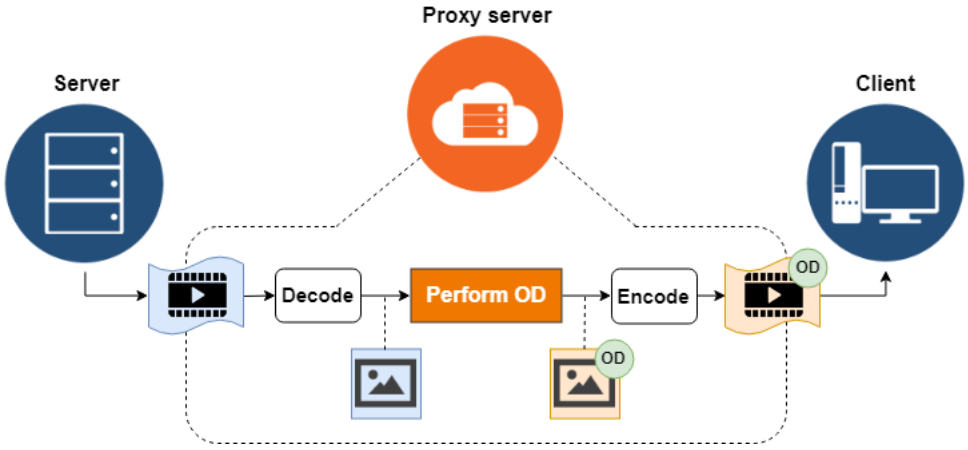


Figure 3.2: Design of proxy server

The proxy server can be implemented using a server machine, located near client's and server's networks.

3.4.3 Client

The client does not need specific hardware for the purpose. It must be able to show the retrieved streaming content (e.g., using a monitor or the smartphone's display). So, it just needs lightweight software to play the content and to easily connect to the proxy server. The client can be identified as a smartphone or a laptop.

3.5 A way to decrease latency

The solution proposed and actually implemented includes a drawback concerning latency that is increased when compared to an on-device computing design. Luckily, the use of modern NICs enables exploiting of data plane libraries, such as DPDK, in order to speed-up packet forwarding and therefore reduce latency. NFV frameworks with support to DPDK, such as ClickNF, can be exploited to overcome this disadvantage.

Chapter 4

Implementation

In this chapter we focus on the actual implementation of the design of NEAR proposed in chapter 3. A top down approach has been used for a better understanding. Starting with a global picture about NEAR's implementation is shown, it follows a detailed description of each component.

4.1 Updating ClickNF

To have a stable environment and obtain familiarity with it, some aspects of ClickNF have been initially improved. These included an update to the DPDK version and the use of the RSS Hash, already provided by the NIC if using DPDK, to sort packets between CPUs.

4.1.1 Update to DPDK 18.05

ClickNF was initially developed with support to DPDK 17.10, released in October 2017. To have the most recent and supported version, ClickNF has been upgraded to version 18.05, the latest available at the time of starting the project. A wide use of pre-processor directives has been done to achieve backward compatibility with older versions. The new version included some changes in the API. For instance, the ambiguous `rte_eth_dev_count()` function used to iterate over ports has been replaced by `rte_eth_dev_count_avail()` and `rte_eth_dev_count_total()` has been built in order to get the total number of allocated ports (available and not).

4.1.2 Use of RSS Hash to speed-up processing

When DPDK is enabled, ClickNF computes an hash to assign the current packets flow to the correct *receive queue*, in order to achieve scalability and high performance, as already described in section 2.6. The current hash function is simple since it requires few elementary bitwise operations. However, it is computed every time a packet is received, even if it belongs to an existing flow (i.e., a packet with the same hash output has been previously processed).

However, to enhance performance, another method to switch packets among the different receive queues has been implemented in this project. The method takes advantage of the *RSS Hash* that is already computed by the NIC itself when receiving a packet. Indeed, when using DPDK, the NIC is responsible to compute and subsequently store the hashcode in the *mbuf* structure.

In this way, when receiving packets belonging to the same flow, there is no need to re-compute the *FlowID*, namely the hashcode, since we just need to access the implemented hash field in the mbuf. However it may happen that the hashcode is not found in the mbuf (e.g., if a packet belonging to a new flow is received), thus the DPDK's `rte_softrss_be` function (i.e., the filter function used by DPDK for RSS) must be performed.

In order to allow users to choose whenever using the legacy or the new optimized method, we extended the DPDK configuration variable set including a new parameter, namely *ENABLE_RSS*. It is a boolean parameter: if 0 (false) DPDK executes the legacy method, otherwise (1, true) exploits the new implemented algorithm[1].

4.2 The complete picture

Figure 4.1 shows the high-level implementation of NEAR. The *Proxy Server*, in the middle, is connected towards the *Server* through TCP Client module and towards the *Client* through TCP Server.

Server and client, two lightweight machines, run FFmpeg and VLC Media Player respectively. Proxy server, the most complex device among the three, exploits OpenCV and TensorFlow's API used for the AR/VR Task, while ClickNF utilize FFmpeg's libraries for stream processing. Next sections give a more detailed description of each component.

ClickNF has been chosen as a programmable framework since its ability to provide high-performance implementations of virtual network functions, which helps, for instance, to deliver video flows extracted from the network into existing AI software packages. ClickNF's modular design allows to seamlessly reuse the majority of the networking code, facilitating the rapid development of the required video processing functions.

The choice of TensorFlow[1] among other AI frameworks (including PyTorch[44], Caffe[29] and Microsoft Cognitive Toolkit[50]) is mainly due the high support given by the developer community in forums and Q&A sites. Moreover, TensorFlow is the framework that better integrates with other computer-vision libraries, including OpenCV[11].

[1]For backward compatibility, 0 (false) is the default value

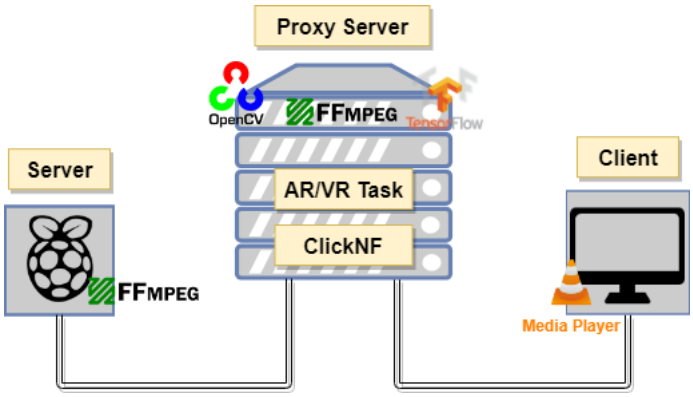


Figure 4.1: High-level implementation of NEAR

4.3 Server

The server is the most lightweight device among the three. It is a Raspberry Pi [46] with a Pi Camera module built on it running two FFmpeg processes, *ffmpeg* and *ffserver*. The first is needed to bind the input source media to a *feed*. That feed is after read by a ffserver process listening on the specified port. In this way, ffserver is ready to stream the media, bound to the selected input source, to clients that request it. Commands and the ffserver configuration file used are listed in listings 4.1 and 4.2.

Listing 4.1: ffmpeg command line. Input can be PI Camera or a file

```
# Input is PI Camera
sudo ffmpeg -ar -f video4linux2 -i /dev/video0 -an http://localhost
    :8090/feed1.ffm

# Input is a media file
sudo ffmpeg -ar -i /path/to/file -an http://localhost:8090/feed1.ffm
```

As the reader can see, the media stream, processed by FFmpeg commands, can be manipulated, e.g., modifying video size, framerate and codec.

Listing 4.2: ffserver sample configuration file

```
HTTPPort 8090
HTTPBindAddress 0.0.0.0

<Feed feed1.ffm>
   File /tmp/feed1.ffm
   FileMaxSize 50M
</Feed>

<Stream video.mpg>
   Feed feed1.ffm
   Format mpeg
   NoAudio
   VideoCodec mpeg1video
```

```
14    VideoSize 640x360
15    VideoFrameRate 30
16    AVOptionVideo flags +global_header
17    # Set qmin and qmax to tune video quality
18    AVOptionVideo qmin 2
19    AVOptionVideo qmax 5
20    PreRoll 5
21    StartSendOnKey
22    VideoBitRate 800
23 </Stream>
```

A basic ffserver configuration file is usually divided into three sections. The first one defines global parameters used for networking, such as IP address and Port to use, maximum allowed bandwidth, max number of alive clients, etc. The second section, identified by the `<Feed>` tag, specifies low-level configuration for the streaming media, including filesystem location and size of the streaming buffer. A third section, identified by the `<Stream>` configures a previously defined *Feed* with media-related parameters. Those include audio and video codecs, optimization parameters and codec-specific flags.

Using the shown configuration file, ffserver instantiate an HTTP server, listening for GET requests at port 8090. When a request is received, if there client has specified an existing media in the URL, ffserver starts sending the streaming media (video.mpg), encapsulated over HTTP.

It should be noted that options *qmin* and *qmax* indicate the *quality range* used for encoding. We experienced very bad video quality with default settings (10 and 42 respectively), hence we experimentally set the range to [2,5] resulting in a negligible quality loss for human eye.

4.3.1 Streaming protocol

This section illustrates the streaming protocol chosen and its functioning. *HTTP streaming* has been used, due to its simplicity and compatibility with most of clients[2].

Figure 4.2 shows HTTP Streaming protocol between a client and a server. Of course, when the Proxy server is used, the request is sent from Client to Proxy server and after from Proxy server to Server.

When the TCP connection is established, client sends an HTTP GET Request specifying the URL of the media to retrieve (e.g., *http://10.0.0.10:8090/video.mpg).* Server checks if that media is available and sends a *200 OK* HTTP message including the beginning of the stream[3].

The HTTP Response header can contain useful information. For instance, *Content-Type* field can be used to infer the codec used by FFmpeg. This is useful when another

[2]Other streaming protocols have been tested, including RTSP and RTMP, however HTTP streaming was the only compatible with the set of chosen frameworks, namely FFmpeg, OpenCV and VLC.

[3]Sometimes, depending on app/socket implementation, the first packet is just the HTTP Response and data is sent starting from the 2nd packet.

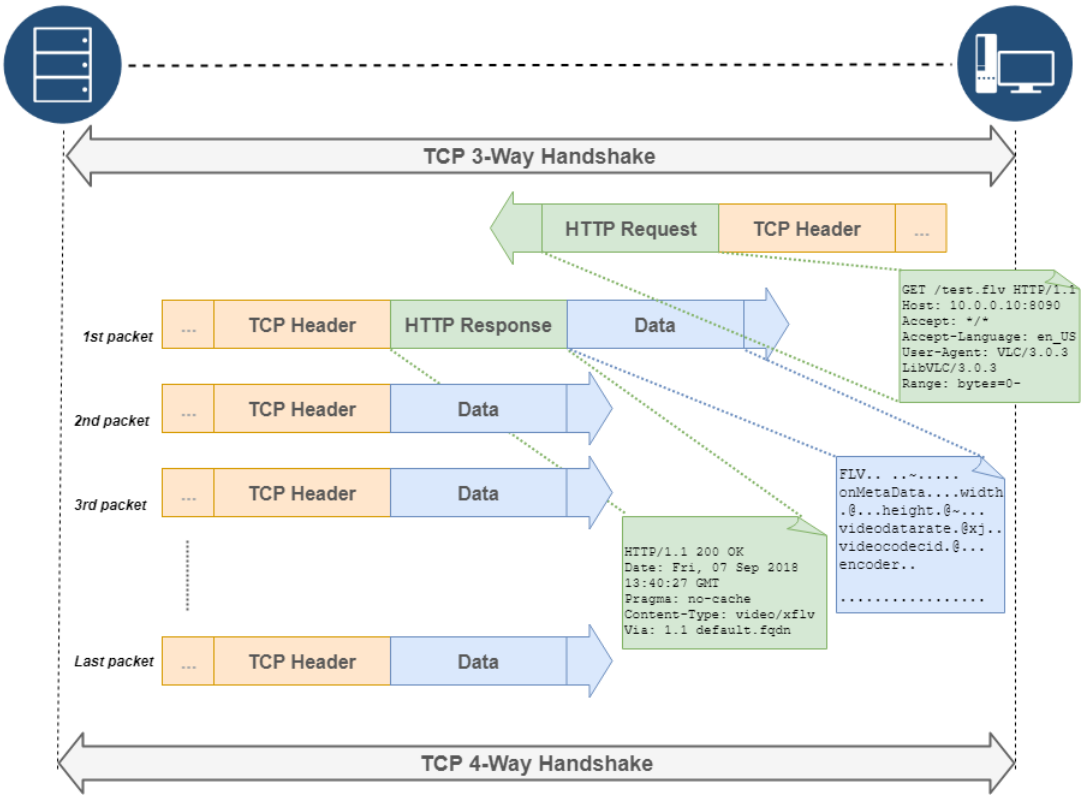


Figure 4.2: HTTP Streaming protocol. Client (right) requests a stream to Server (left)

application wants to re-encode the streaming media using the same codec.

The data field contains the actual media information encapsulated in the specific media container. Data can include part of, one or multiple encoded frame(s), depending on the media size (e.g., resolution and/or quality). It has to be noted that following frames are segmented and directly encapsulated over TCP.

4.4 Proxy server

Figure 4.3 illustrates the skeleton of the proxy server. The latter runs two processes, the AR/VR Task (Object Detection) and ClickNF. The new elements implemented are green-highlighted and these include SOCKS 5 Proxy, App Classifier, Decoder, Encoder and Object Detection. Those elements are described in details in the following sections.

Object Detection has been implemented outside the ClickNF framework for two main reasons. First, it is a complex task, that requires a wide set of libraries (OpenCV, TensorFlow, Threads, ...) that ClickNF does not need. Normally, to run a TensorFlow application for inference, it is needed to compile using Bazel [17], a Google's buildsystem. Of course, this is a problem when dealing with other projects that are built using other buildsystems (e.g., Autotools for ClikNF). As a workaround, we have built a shared TensorFlow library that can be easily distributed across multiple compatible platforms and

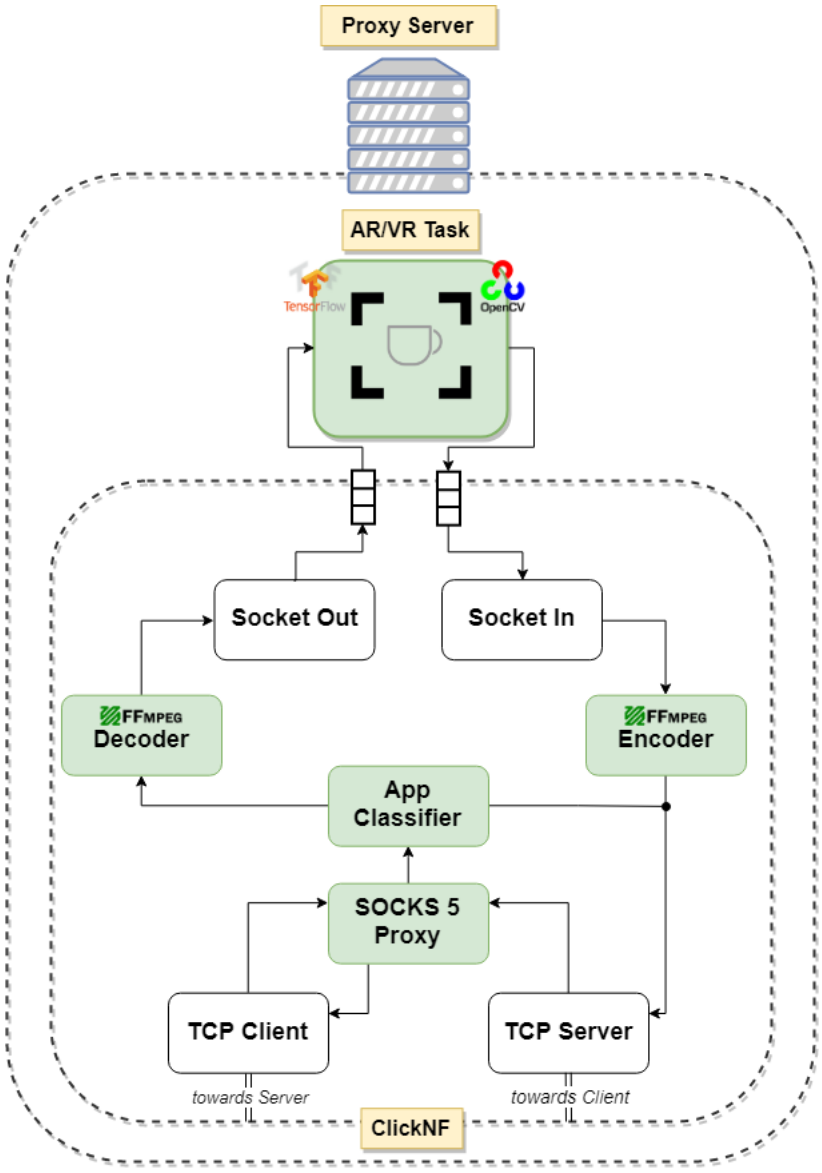


Figure 4.3: Proxy implementation

installed through common buildsystems (Makefile, CMake, ...). This can ease, in a future work, the migration process of AR/VR Task into ClickNF[4].

The second reason is due to the *Run-To-Completion* scheduling model used by Click (and ClickNF). In fact, the AR/VR Task could spend a lot of time to perform its job. This would deny the scheduling of other tasks, which can result in poor performance or in the worst case a framework deadlock.

[4]Not for nothing, Object Detection has been implemented using the C/C++ language, the same used for ClickNF.

Hence, to allow the two processes to communicate each other, a IPC communication is needed. Since message granularity is at packet level (we receive TCP packets from the streaming server) it would be convenient to use Inter-Process Communication (IPC) mechanisms that can work using the same. Generally, the only solution is using network sockets in localhost. However, a Unix environment offers also a high-performance type of sockets, called Unix Domain Sockets. A result by Redis [47] displayed in Figure 4.4 shows that Unix Sockets can achieve around 50% more throughput than TCP Sockets in localhost.

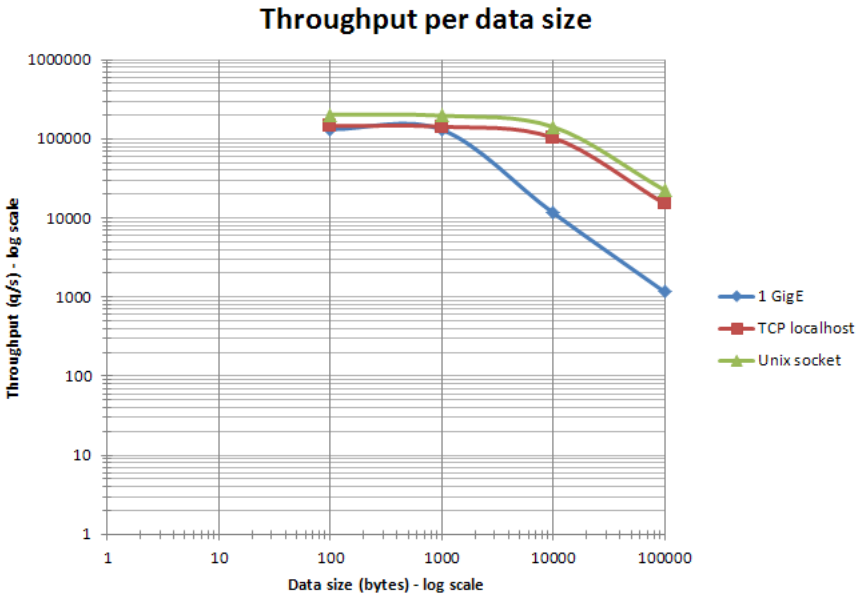


Figure 4.4: Redis Throughput benchmark: Unix Sockets vs TCP localhost

4.4.1 SOCKS 5 (Lite) Proxy

The use of VLC as a media player running on a client in order to read the streaming video trough a proxy server required an implementation of SOCKS 5 proxy server. SOCKS has been designed to allow incoming/outgoing traffic from/to a corporate LAN trough a firewall, thus achieving the so called *"Firewall traversal"*. However, for the aim of the project, SOCKS is exploited in order to have an easy way to retrieve a live streaming media through a middleware, where ML algorithms can be performed, rather than directly from a streaming device.

ClickNF's set of Elements already includes a SOCKS 4 proxy server implementation but the backward incompatibility of SOCKS 5 makes it unfeasible to use. SOCKS 5 [27] introduces new features (including UDP support and authentication) that makes it harder to implement. Thus, for the purpose of the project, a simplified SOCKS 5 proxy server compatible with a VLC client has been built.

The SOCKS 5 Proxy element (cf. fig. 4.3), working in push mode, has two input port (from TCP Client and TCP server) and two output ports (towards TCP Client

and App Classifier). This asymmetry comes from the reason that packets arriving from Server must be switched to the correct element (*Decoder* or TCP Server depending if they contain data). Hence, all packets retrieved by Server must first be classified (see section 4.4.2). Figure 4.5 depicts the implemented SOCKS protocol.

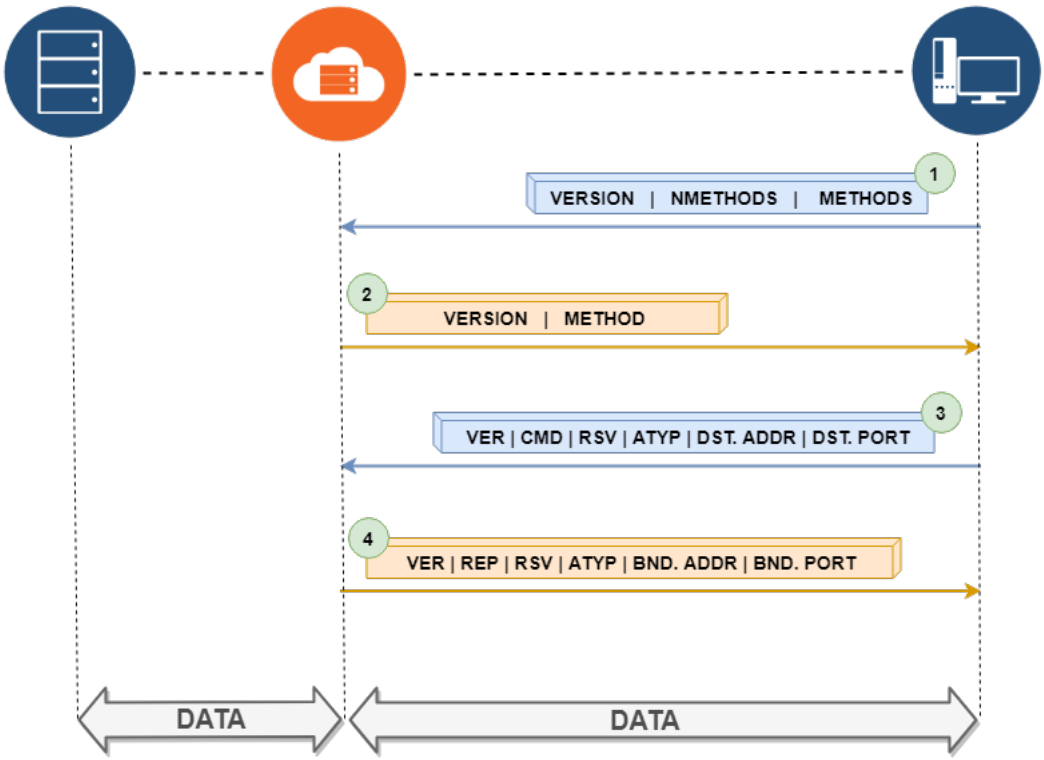


Figure 4.5: Socks 5 Proxy

1. The client connects to the proxy server, and sends a *version identifier/method selection* message. It specifies the supported version followed by the number and the list of supported methods for authentication.

2. The proxy server selects from one of the methods given in *METHODS* (*NOAUTH* in this implementation), and sends a *method selection* message to announce version and method chosen. Those first two messages, introduced in version 5, are used to negotiate the features supported by client and proxy server.

3. The client sends a *proxy request*. The command (*CMD*) is *CONNECT* but can also be *BIND* or *UDP ASSOCIATE* [5]. In this implementation, *ATYPE* (Address Type)

[5]CONNECT is used when basic proxying is needed. BIND is used in protocols where the client must accept connection from server. UDP ASSOCIATE purpose is to establish an association within the UDP relay process to handle UDP datagrams.

is “IPv4”, meaning the address specified in DST.ADDR is an IPv4 address in dotted decimal notation. DST.ADDR and DST.PORT are respectively the address and the port of the server to connect to.

4. The server answers with a *proxy reply*. Here, ATYP indicates that *BND.ADDR* is an IPv4 address in network byte order [6].

Finally, after the handshake, data can be exchanged between client and server though the proxy server.

Our SOCKS 5 implementation is compliant with L7 protocols. A Click annotation (2.4.1) *“SOCKS_APP_FLAG_ANNO”* is used to label the packet as application-layer data before being forwarded to the next element. It means that packet must be sent to upper layers and not directly to the other endpoint (i.e., the client).

4.4.2 App Classifier

This simple element is needed to forward packets in the right way. It has one input and two output ports, 0 and 1. When a packet comes in the input port, App Classifier looks at packet's annotations. If *SOCKS_APP_FLAG_ANNO* is present, meaning that packet contains data to be processed by upper layers, the packet is forwarded to port 1[7], 0 otherwise. Moreover, if the incoming packet contains *TCP_DEL_FLAG_ANNO*, because server requested to close connection, App Classifier forwards that packet to port 1. The algorithm is shown in appendix A.1.

4.4.3 Decoder

The *Decoder* element, configured to work in push mode, has one input and one output port. It is a stateful component, meaning that its execution depends on the current state of the element. The possible consecutive states are the following.

- **HTTP_PARSE:** It is the initial state. *Decoder* is waiting to receive the first Click packet in order to send the HTTP header. During this state *Decoder* strips out the HTTP header and makes it available for other elements (i.e., Encoder) through shared data structures. At the end of this state, the HTTP header is internally store.
- **INITIALIZE:** *Decoder* is waiting to receive the first data packet. It is useful to initialize FFmpeg's libraries only once.
- **DECODING:** Used when decoding subsequent frames. It is the state where *Decoder* spends most of its time, since following packets enter this state.

[6] According to RFC 1928, BND.ADDR and BND.PORT are the IP address and PORT used by the proxy server to connect to the server. In this implementation IPv4 address *0.0.0.0* and port *0* are used for convenience. Obviously, those values are different from the ones actually used. Nevertheless, most of clients ignore those values.

[7] Defined to be the port to the upper layer element.

- **LAST:** It is the final state. *Decoder* enters it when sending the last frame of a stream.

The complete algorithm is shown in appendix A.2.

HTTP_PARSE

Initially, the first packet forwarded to the *Decoder* element should contain the whole HTTP header, as described in section 4.3.1. Since the HTTP response may be split in several packets, depending on its size, a *buffered input* has been implemented. This means that *Decoder* is able to wait for a specific amount of packets in order to retrieve the entire HTTP response, even if it is split up into more TCP packets. Moreover, *Decoder* can manage the case one packet contains both HTTP response and Data that are stored using different buffers. At the end of this phase, The HTTP response is stored in an internal buffer, shared with and accessible by external elements (including *Encoder*), and the state is set to *INITIALIZE*.

INITIALIZE

At this point, *Decoder* waits to receive a configured number of packets (N) before starting to decode. Those packets are inserted into a *PacketQueue* structure and processed when reaching the size of N. This workaround is needed to make the decoding function properly work. In fact, we experienced an anomalous behavior due to FFmpeg APIs when reading packets from memory in order to feed the decode function. In this scenario, FFmpeg's *av_read_frame* continuously [8] reads bytes from memory even when no data is available, returning an error and forcing the decode function to be in an inconsistent state. We noticed that reading from file is not a problem, since data is always available.

Thus a way to solve such problem is to call the decode function when PacketQueue contains sufficient packets. This threshold highly depends on the media. Parameters needed to appropriately tune the threshold are above all video resolution and audio/video codec. The higher is the video resolution, the higher must be the threshold (*QUEUE_SIZE*). Table 4.1 shows a possible setup when using FLV and MPEG codecs.

Resolution	**360p**	**720p**	**1080p**
QUEUE_SIZE	64	128	256

Table 4.1: Threshold values for a correct decoding functioning

After that the threshold is reached, FFmpeg's elements are properly initialized, if not done yet. The FFmpeg function `av_probe_input` is internally exploited to guess the input format (i.e, the codecs) of the stream. This run-time initialization allows to dynamically and automatically configure *Decoder*. In this case the appropriate codec to use is automatically chosen after probing the first packet received.

[8] up to six times.

DECODING

After initialization, *Decoder* enters the state *DECODING*. This is the where *Decoder* spends most of its time, since following calls to *Decoder* enter directly this state. When in *DECODING* state, if the aforementioned threshold has been reached, the decode function is actually performed, otherwise *Decoder* returns and waits for other packets. Moreover, in order to stabilize PacketQueue's size, when decoding is finished, the decode function is called again if PacketQueue's size still exceeds the threshold.

A callback function `readPacket` is used in order to read an encoded frame when needed. The callback is registered using FFmpeg's APIs and internally called by `av_read_frame`. The latter can run the callback multiple times depending on the amount of packets needed to retrieve a frame. When `readPacket` is called, it continuously pops packets from *PacketQueue* and stores them in a buffer shared with FFmpeg until the total number of requested bytes is reached.

When packet(s) is (are) decoded, if a frame is released, it is first converted from YUP420P to BGR24 [9] and after sent to the next element, using port 0. The frame is fragmented into more packets, since DPDK packets are limited by size. User can specify that packet size in a Click configuration file.

Note that to let *Encoder* know video resolution, frame dimensions (width and height) are sent using a packet with 4 bytes (2 for each dimension) before sending the first packet (belonging to the first frame of the stream). Those dimensions are valid for the whole frame set since images extracted from the same video have the same resolution. Hence, that 4-bytes representing video resolution are sent just once initially, and must be correctly read at receiving-side before reading the actual frame data.

LAST

Finally, if the streaming media is going to finish or user asked to terminate the session, *Decoder* is notified with an empty packet containing the annotation *TCP_DEL_FLAG_ANNO*. *Decoder* enters the state *LAST* and flushes all queued packets. However, Decoder cannot brutally close connection (i.e., send that annotation to TCP Server) because there could be still frame processing at Encoder or in the AR/VR Task that must be sent before closing connection. To avoid that, *Decoder* registers the number of decoded frame so far and let later *Encoder* close connection using a mechanism that will be described in section 4.4.6.

Configuration parameters

Table 4.2 defines the possible parameters that can be set-up with a Click configuration file.

[9] BGR24 is easier to manage with OpenCV.

Name	Description	Values	Default
VERBOSE	Print verbose output	True, False	False
PKT_MAX_SIZE	Packets maximum size that *Decoder* uses for fragmentation	> 0	1448
QUEUE_SIZE	Threshold to use for *PacketQueue*	> 0	64

Table 4.2: *Decoder* configuration parameters

4.4.4 SocketIn and SocketOut

The Click framework already provides a *Socket* element that can be configured to implement stream, datagram or unix domain sockets. The desired mode of operation is achieved by reading specific parameters from the Click configuration file. For instance, given the configuration file in listing 4.3, Click setups a tcp client (S1) connecting to server 10.0.0.1 trough port 80. In the same way (S2) is a tcp server listening at port 80 on all interfaces.

When using unix sockets, the communication channel is identified by a filesystem path. In fact, the unix socket is a file buffer that is usually read/write by two different processes. In this scenario, client (SocketOut) is configured to send received packets to `./socketX`, while server (SocketIn) waits for a connection from a client and reads packets from `./socketY` when available.

Listing 4.3: Click declaration for Socket elements

```
1   # TCP client
2   S1 = Socket::(CLIENT true, TCP, IP 10.0.0.1, PORTNUMBER 80);
3
4   # TCP server
5   S2 = Socket::(CLIENT false, TCP, IP 0.0.0.0, PORTNUMBER 80);
6
7   # UNIX client
8   SocketOut = Socket::(CLIENT true, UNIX, FILENAME ``./socketX'');
9
10  # UNIX server
11  SocketIn = Socket::(CLIENT false, UNIX, FILENAME ``./socketY'');
```

Concerning the server-side *Socket* implementation, it has been built to be portable and modular. It means that *Socket* works for any client/server framework. However, having such modularity and portability sometimes make even more harder the development of upper-layer elements exploiting *Socket*. To be more clean, when building a common stream-oriented client-server application, both the endpoints know the protocol and format they are using, that is, server knows exactly *how* and *how much* read. On the other hand, when including a third party (*Socket* in our case), the latter does not know protocol and format used between the two endpoints (AR/VR Task and Encoder/Decoder).

For a better understanding, consider the scenario proposed in fig. 4.6 where a client sends two frame via socket. The picture illustrates the implementation of two servers. The first one (*Server 1*) is really straightforward, since it uses simple APIs to retrieve

exactly a specified number of bytes (*nread*). On the other hand, *Server 2* is composed by two elements, Socket and a Encoder, and its implementation is much more complex. *Socket* server reads from socket up to a fixed number of bytes (1448 by default) and directly sends to next element as much data as it can (can be less than 1448, depending on kernel behavior). Next element (i.e., *Encoder*) receives a packet, that is, part of a frame. So, it must enqueue all the received packets in a buffer, in order to finally build a frame. Moreover, it must consider the case the receive packet contains parts of different frames. To deal with this problems, *data buffering* must be performed by upper-layer elements exploiting *Socket*, as later described in section 4.4.6.

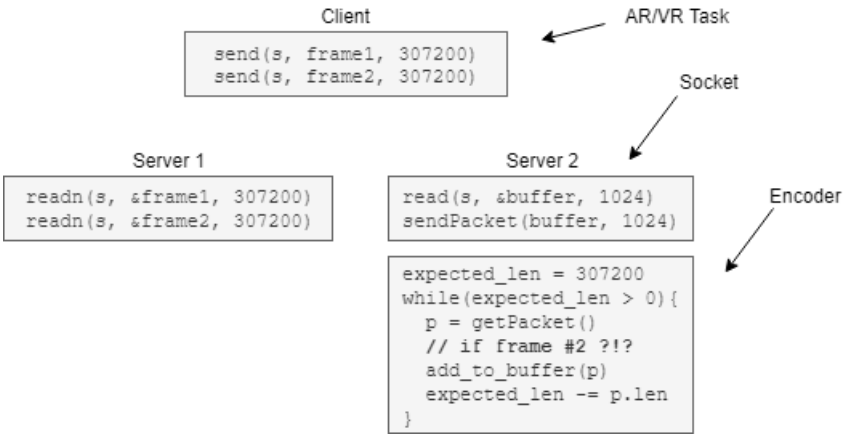


Figure 4.6: Socket server implementation comparison

4.4.5 AR/VR Task: Object Detection

Object Detection (*OD*) is the implemented AR/VR task[10] that exploits TensorFlow's libraries to detect multiple objects (such as keyboards, monitors, people, cats, dogs, etc...) in the same image. Each object is identified using a label with a specific score. The latter, a decimal point number between 0 and 1, defines the *accuracy* (i.e., how sure is the algorithm about the correct identification). A snapshot of an instance of *OD* is shown in fig. 4.7, where different objects such as keyboards and monitors are detected according to the aforementioned style. Appendix A.3 describes the high-level algorithm of the implemented AR/VR task while next sections present a more detailed functioning.

Basic functioning and dual-thread approach

The AR/VR Task works outside ClickNF, thus, in a separate process. To make it work together with ClickNF, an IPC mechanism (e.g., Unix Domain Sockets) is needed as previously described in section 4.4. Two unix sockets are used, one (*SockIn* for data input

[10]It is actually an AR task, due to the lack of interactivity from the user.

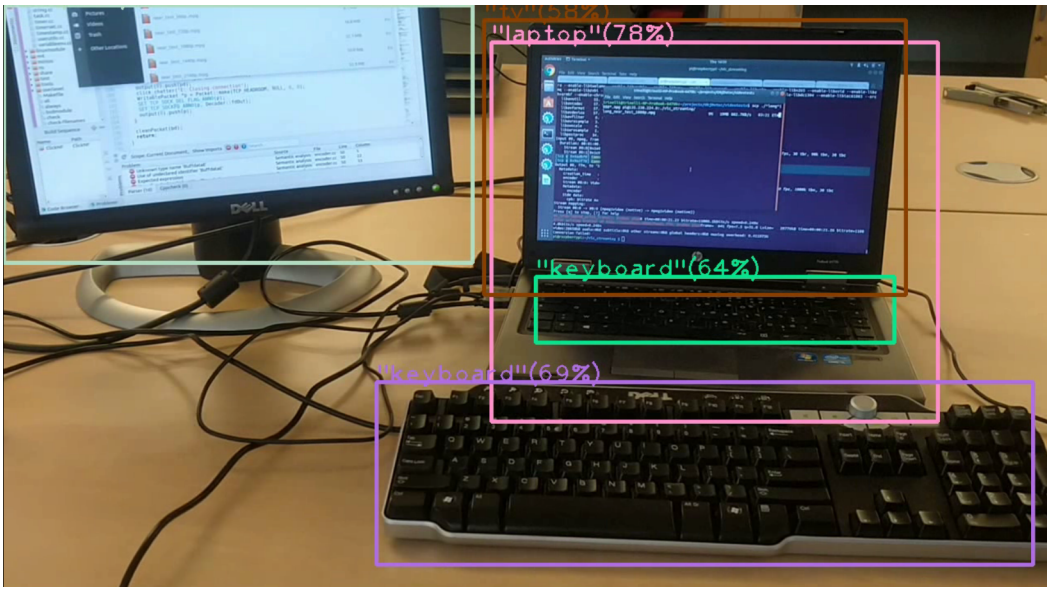


Figure 4.7: Snapshot of Object Detection

and the other *SockOut* for data output[11]). The basic functioning of *OD* is straightforward. *OD* receives a frame trough *SockIn* from ClickNF, performs *OD* and sends back the augmented frame to ClickNF.

However, in order to achieve better performance (i.e., shorten latency time) the actual implementation is more detailed. Rather than working with a single frame at a time, *OD* can retrieve frames while running TensorFlow and sending out frame. To achieve such functionality, two threads working in a producer-consumer fashion way are used. A thread (*Receiving thread*), working as the producer, continuously reads frames from *SockIn* and inserts them into a queue shared with the other thread. The latter (*AR thread*), whenever a frame is available in the queue, gets a frame from the queue, performs Object Detection and sends the augmented frame through *SockOut*. See fig. 4.8 for a better understanding.

Furthermore, in order to synchronize producer with consumer (i.e., informing consumer that data is available), avoid race conditions and more generally make the shared queue thread-safe, we used a *condition variable* working with a *mutex* (i.e., it means using a *monitor* [22]).

To avoid secondary race conditions we used some C++ advanced utilities for thread safety exploiting a *RAII*[12] paradigm. In fact, the mutex is wrapped by a *lock_guard* producer-side, and by a *unique_lock* consumer-side. The latter has the same behavior of *lock_guard*[13], with the exception that can release the lock before destruction. In our

[11]Note that naming is inverted compared to the one used in Click, since output for Click is input for AR/VR Task and vice-versa.

[12]Resource Acquisition Is Initialization

[13]Lock and unlock happen respectively during construction and destruction of the object itself.

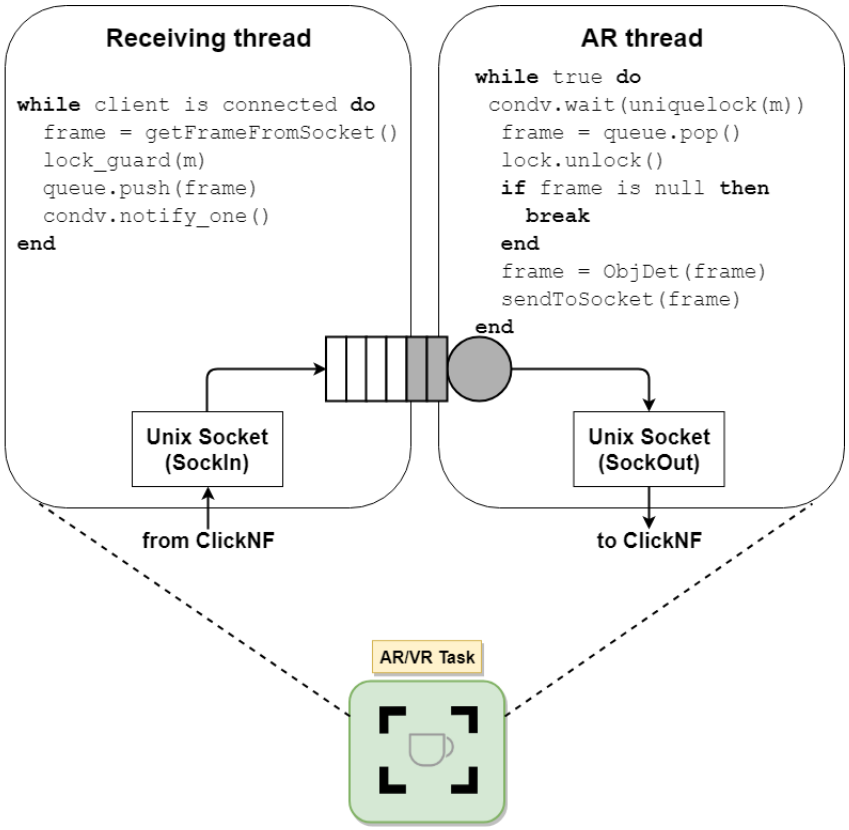


Figure 4.8: Implementation of the AR/VR task: Object Detection

case, this is useful since we only need to make safety the popping from the queue, not the whole TensorFlow's Object Detection.

Finally, `getFrameFromSocket()` returns a null pointer if no frame is available. This means client (ClickNF's Socket element) has closed connection, namely the stream is at the end. Thus, the Receiving thread inserts the null pointer in the queue and closes. The AR thread, when retrieving the null pointer breaks its loop and terminates.

The core of Object Detection

To run the OD task an existing trained model has been used for the purpose. Thus, we worked on the inference part of a Deep Neural Network algorithm rather than on training. The COCO-trained model `ssd_mobilenet_v1_0.75_depth_coco` [16], one of the fastest having acceptable accuracy, has been chosen to allow the comparison of NEAR with an on-device design, as later shown in section 5.3.1.

The actual TensorFlow's Object Detection task is performed by the *AR thread*. First of all, it initializes TensorFlow's model and loads the dictionary of labels. Of course, the dictionary must be consistent with (i.e., equal to) the one used for training. After that, for each retrieved frame from the shared queue it, first the *input tensors* are extracted from the frame and after that a *TensorFlow session* is launched with retrieved tensors as

input.

To increase performances, the session internally exploits multi-threading using two pools of threads, namely `intra_op_parallelism_threads` and `inter_op_parallelism_threads`, configurable by command line arguments. The first set is used when the execution of an *operation* can be internally parallelized. The second set is used when multiple *nodes* performs independent operations. By default, TensorFlow assigns to the two pool a size equal to the number of total cores in the system, but we have experienced better performance with other numbers, as later shown in section 5.3.1.

Moreover, note that it is not actually necessary to perform *OD* for each received frame. In fact, the images we are considering have been extracted from a video thus skipping the detection on some frames would have a negligible effect to the human eye. Hence, the session actually runs once every N-times, where N is the *step* chosen by the user. We call *"skipped frames"* those where *OD* is not applied. Anyway, in order to avoid abnormal video artifacts, skipped frames are not completely ignored. In fact, detection-related statistics of a frame where *OD* is actually applied are kept for the following N-1 (skipped) frames. This feature dramatically increases performance as shown in section 5.3.2.

Finally, the session produces four output tensors, namely **boxes**, **scores**, **classes** and **num_detections**. *Boxes* contains the coordinates of the two points (upper-left and lower-right) that represent the rectangle surrounding the detected object. *Scores* represents the detection accuracy, as a decimal point number between 0 and 1. *Classes* includes the *ID* of the identified item used as index to access the dictionary of labels. *num_detection* is an integer specifying how many objects TensorFlow was able to detect in the frame.

Drawing results

As last step, the algorithm draws the *num_detection* boxes, skipping detections with low score, that is, under a threshold specified by command line arguments[14]. The drawing of the results is made exploiting OpenCV's utilities including `cv::rectangle`, to draw the box given the *boxes* tensor, and `cv::putText` to write down label and accuracy, given *classes* and *scores* output tensors.

CPU only and GPU-accelerated modes

OD has been designed to work in two modes: *CPU-only* and *GPU-accelerated*. Using the former, *OD* exploits the two TensorFlow's pools of threads as previously mentioned. The latter goes beyond, by adding GPU support, resulting in dispatching computation among both CPU (thus using TensorFlow's threads allocated in the CPUs' cores) and GPU. This mode generally results in increased performance but may have counter-effects when allocating a high-number of threads (cf. section 5.3.1).

[14]Default is 0.5.

4.4.6 Encoder

The *Encoder* element is configured in push mode, with one input and one output port. As already mentioned, *Encoder* must co-operate with *Decoder*, for instance, to synchronize about the last frame decoded and to retrieve the HTTP header owned by *Decoder*. Instead of using a communication channel, that would need one more port both in *Decoder* and *Encoder* and a notification protocol, some *Decoder*'s properties have been made static and public, thus accessible from external components, like *Encoder*. The complete algorithm is presented in appendix A.4.

The possible consecutive states of *Encoder* are the following.

- **SEND_HTTP:** It is the initial state. *Encoder* retrieves the HTTP Response header stored in *Decoder* and forwards it to the next element.
- **INITIALIZE:** *Encoder* is receiving packets belonging to the first frame. At the end of this state, FFmpeg's structures are initialized.
- **ENCODING:** *Encoder* waits to receive a complete frame (using data buffering) that is finally encoded using FFmpeg's APIs. Following calls to *Encoder* enter this state.
- **LAST:** It is the final state. *Encoder* understands the encoded frame is the last of the stream and closes connection.

As previously mentioned, a counter-effect of using a *Socket* element is that the number of bytes read through socket are fixed and specified at configuration time. In fact, it returns as much data as it can read, but the read is limited by a previously configured buffer size[15]. However, a frame is usually bigger than 1448 bytes, moreover, it may happen that *Packets* having length less than 1448 are received due to socket implementation. To deal with this inconvenience, data buffering is needed.

As basic functioning, *Encoder* copies the incoming packet in a buffer (*frame buffer)* used by FFmpeg's API to encode a frame. More specifically, based on the reading state *Encoder* is, it must copy (part of) data from the buffer to its own variables. To clarify, consider for example that *Encoder* receives a packet 'p' having 1448 bytes, where the first 1024 are the last part of frame 'n' while the 24 bytes left belong to frame 'n+1'. In this case, *Encoder* is able to copy the first 1024 bytes to the buffer (that contains bytes related to frame n) and the last 24 bytes to another temporary buffer. When packet 'p+1' is received by *Encoder* , the content of the temporary buffer is first copied to the frame buffer, after 'p+1' content is copied.

SEND_HTTP

When *Encoder*'s push method is called for the first time, being at *SEND_HTTP* state, it retrieves the HTTP header accessing *Decoder*'s class and pushes it to port 0. Thus, it will be forwarded to the client, through TCP Server. Moreover, being the first packet, it retrieves video's width and height, that will be later used for encoding.

[15] In other words, it can read a certain number of bytes less or equal the buffer size specified during configuration. Default is 1448.

INITIALIZE and ENCODING

Received a complete frame, the *encode* function is called. If called for the first time (i.e., the first frame has been received), it initializes the needed FFmpeg objects and retrieves the used codec from *Decoder*. After that, it performs FFmpeg's encoding, and if a *packet* is retrieved, it is properly written using `av_write_frame` and forwarded to the next element. Specifically, *Encoder* uses a callback named `writePacket`, internally called by `av_write_frame`, used to write in memory an encoded packet. *Encoder* has a reference to that area of memory and available packets are therefore forwarded to the next element. Note that initialization must occur only once: following calls to *Encoder* enter the state *Encoding*.

LAST

Encoder and *Decoder* keep their own frame counter[16]. When such frame counters are equal *Decoder*'s status is set to *LAST*, it means *Encoder* has receives the last frame for that stream, thus connection must be closed. Hence, *Encoder* first sends the last packet, using `av_write_frame` and finally writes the proper trailer, with `av_write_trailer` including the *TCP_DEL_FLAG_ANNO* annotation to announce to *TCP Server* to close connection.

Configuration parameters

Table 4.3 defines the possible parameters to use with *Encoder* in a Click configuration file.

Name	Description	Values	Default
VERBOSE	Print verbose output	True/1, False/0	False
CODEC	Codec to use for encoding	mpeg, flv, ...	M [17]
QMIN	Lower bound for quality adjustment/sampling	> 1	10
QMAX	Upper bound for quality adjustment/sampling	$>$ QMIN	42
PKT_MAX_SIZE	Packets maximum size that *Encoder* uses for fragmentation.	> 0	1448

Table 4.3: *Encoder* configuration parameters

4.5 Client

The client is a lightweight device that must be able to retrieve and display a streaming media. To be compatible with the Proxy server, the media player must implement

[16] *Decoder* keeps how many frames has sent and *Encoder* how many received.

[17] Mandatory

a SOCKS 4/5 client. VLC is a good, well-known and easy-to-use media player. The choice of using such media player is due to the user-friendly configuration setup when using SOCKS. Moreover, it provides both CLI, with the command `cvlc`, and GUI (using `vlc`).

For instance, to retrieve the a streaming media with Object Detection the client must simply run the second command shown in listing 4.4.

Listing 4.4: VLC command line to play the streaming media

```
# connect to streaming server having IP 10.0.0.10, listening on port
    8090 and streaming a MPEG1 video identified as video.mpg.
vlc http://10.0.0.10:8090/video.mpg

# same as before, but pass through a SOCKS proxy server having IP
    address 10.0.0.100 listening on port 9000.
vlc --socks 10.0.0.100:9000 http://10.0.0.10:8090/video.mpg
```

Using the second command line, the proxy server, already configured to run Object Detection, retrieves the native streaming media from the server and send the augmented media to the client, as previously described in section 3.4. The output was shown in fig. 4.7.

4.5.1 A note about SOCKS and transparency

NEAR's implementation allows the use of AI-incompatible device in order to actually deploy AR/VR services. This is done with without installing any additional software server-side and with almost zero-impact on client. Those are the reasons why NEAR is defined to be *transparent*.

To achieve a full-fledged transparency, meaning that no modifications are required both client and server side, advanced network analysis techniques such as Deep Packet Inspection [7] and classification might be exploited at the edge network.

On the other hand, we have chosen to use SOCKS since already implemented in ClickNF[18] and because it is highly-supported by most of media players. Nevertheless, it has been ported to numerous hardware/software platforms.

[18]SOCKS v5 required light modifications compared to SOCKS v4.

Chapter 5

Results

This chapter presents interesting results regarding different experiments. First, we argument the changes on performance when using the implemented RSS hash for optimized dispatching of packets among different cores. The other two experiments show the advantages of using NEAR rather than an on-device approach.

5.1 Testbed

The framework used for implementation purposes presented in fig. 5.1 is composed by a Raspberry Pi and two server machines, namely *Freisa* and *Barolo*, acting as proxy server and client respectively. Note that the three devices are all located in the same LAN. In this way, for testing purpose, client can directly connect to server without proxying. To display the VLC media player output, *Barolo* is attached to a LCD screen. Finally, in order to perform fast packet forwarding, *Freisa*'s interfaces *eth1* and *eth2* run DPDK libraries.

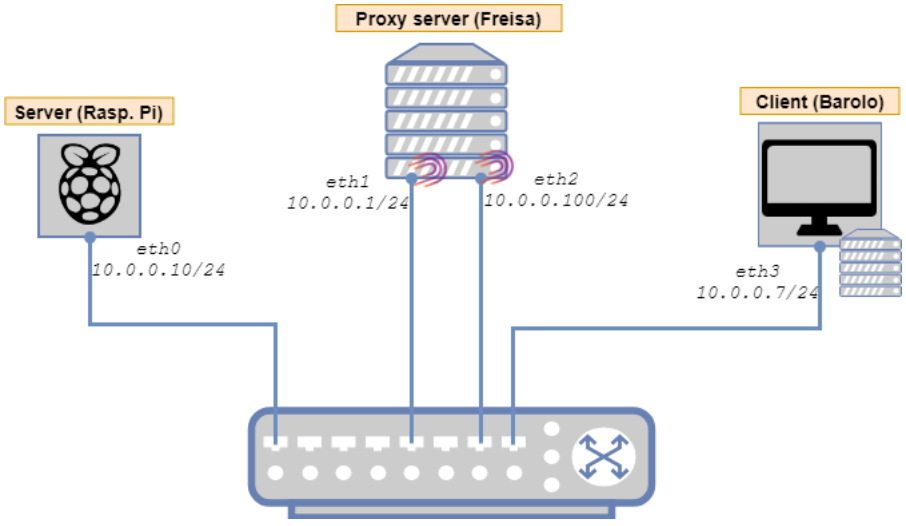


Figure 5.1: Testbed

5.1.1 Specifications

Table 5.1 shows the characteristics of the machines used in our testbed. Note that *Freisa* and *Barolo* are two identical machines, with the difference that the former is equipped with an NVIDIA GPU card and the latter has a LCD display attached. Both systems are equipped with 2 CPUs, having 10 physical cores each one in *HyperThreading* mode, resulting in a total of 40 cores (20 of them are virtual).

For multi-CPU systems (*Freisa* and *Barolo*), since each CPU is linked to its own RAM, a Non-uniform memory access (*NUMA*) is used when a CPU needs to retrieve data stored in the memory of other CPUs. Intel systems provides a point-to-point low-latency interconnection, namely Intel® QuickPath Interconnect (*QPI*) [25], for rapid remote memory access from external CPUs.

	Rasp. Pi	**Freisa**	**Barolo**
Role	Server	Proxy Server	Client
CPU	ARM Cortex-A53 4 cores @ 1.2 GHz	2x Intel Xeon R E5-2660 v3 40 cores @ 2.6 GHz	2x Intel Xeon R E5-2660 v3 40 cores @ 2.6 GHz
RAM	1 GB	64 GB	64 GB
Others	Pi Camera	HyperThreading, NUMA, NVIDIA GeForce GTX 980	HyperThreading, NUMA, LCD display

Table 5.1: Testbed specifications

5.2 RSS Hash

As discussed in section 4.1.2, a new method, namely RSS Hash, has been implemented in order to optimize the dispatching of packets belonging to the same IP flow. Figure 5.2 illustrates the comparison between the legacy and the new proposed algorithm when running an echo server-client experiment. In this experiment, we run an *echo-server* ClickNF application using different number of CPUs, from 1 to 8, and we plot the number of messages per second[1].

We did not experience substantially changes when using our algorithm. A possible reason is that, when using *RSS Mode*, Although caching the hash computation in the mbuf, the hash function used by DPDK, namely `rte_softrss_be`, is more onerous and complex than the one used by Click, being a composition of XORs.

[1]The number of messages per second is given as statistics at the end of a ClickNF echo-client application.

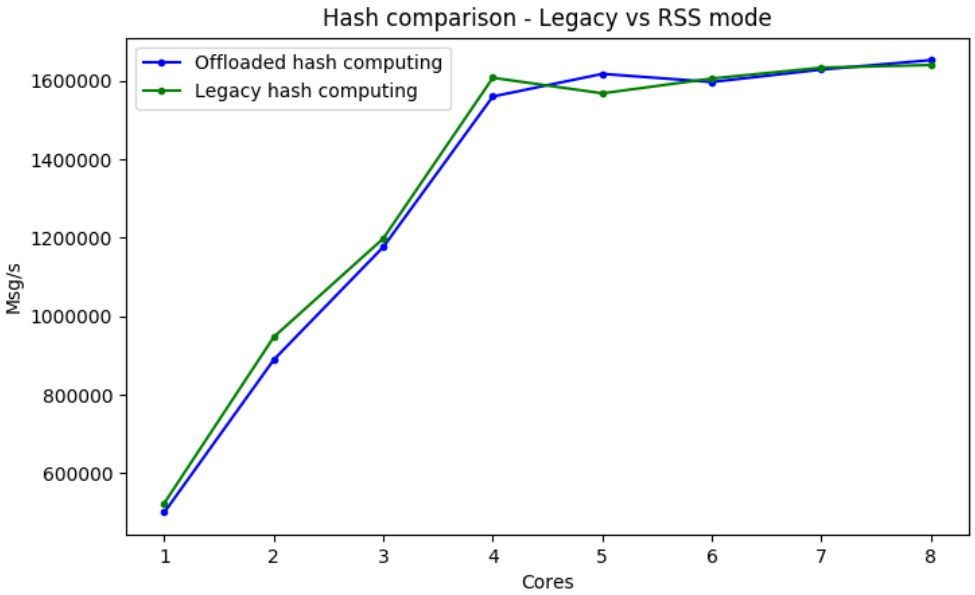


Figure 5.2: Flow ID computation comparison

5.3 On-device vs Edge computing

In this section we compare a scenario where only a legacy device is used to implement an AR/VR service with respect to the use of NEAR. As described in section 3.3, the expectations are that NEAR better performs in terms of computational power, scalability and flexibility at the cost of a slightly increased latency.

5.3.1 FPS comparison

As motivation for the initialization of this project, we wanted to highlight the quality gain due to offloading computing power to the edge infrastructure. In this experiment, quality is measured in terms of Frames Per Seconds (FPS).

Two different devices, namely *Rasp. Pi* and *Freisa*, used the same simplified Python version of *Object Detection*. In this implementation, *OD* continuously extracts frames from an input video, directly from local disk, and performs TensorFlow's Object Detection on each one. For each iteration, a new frame is grabbed, *OD* is performed and boxes and labels are being drawn. This process is repeated for each frame in the input video. In this simplified implementation the augmented frame is simply discarded.

Note that we experienced similar results when comparing the simplified Python version of *OD* with the one used by NEAR (that exploits TensorFlow's C++ APIs). Thus, we can consider the results in fig. 5.3 as being actually achieved when using NEAR.

For each frame i, two clocks are used to retrieve the exact current time. The first clock

(*t1*) registers the time just after a complete frame is grabbed from video, while the second one (*t2*) registers the time immediately after labels and boxes are drawn. The complete formula to compute the *instantaneous FPS (iFPS)* is shown in eq. (5.1).

$$iFPS(i) = \frac{1}{clock_{t1}(i) - clock_{t2}(i)} \tag{5.1}$$

For the experiment, we used a sample video recorded with a smartphone displaying, in one minute, office furnitures (monitors, chairs, desks, laptops, etc.). The video has been recorded in Full-HD (1080p) at 30 FPS. Moreover, for comparison purpose, we converted the input video to 360p at 30 FPS.

Figure 5.3 shows the results of the experiment. Each line is described by the *mean* of the computed FPS(i) for each video source. During the experiment, we launched *OD* using different number of threads to understand what are the optimal values for `intra_op_parallelism_threads` and `inter_op_parallelism_threads`, the two pools of threads previously described in section 4.4.5.

Finally, we enabled GPU-support for NEAR (both using C++ and Python TensorFlow's libraries). GPUs can be used besides CPUs to accelerate computing, improving NEAR in terms of performance and scalability. Figure 5.3 shows a comparison between the CPU-only and GPU-accelerated modes[2].

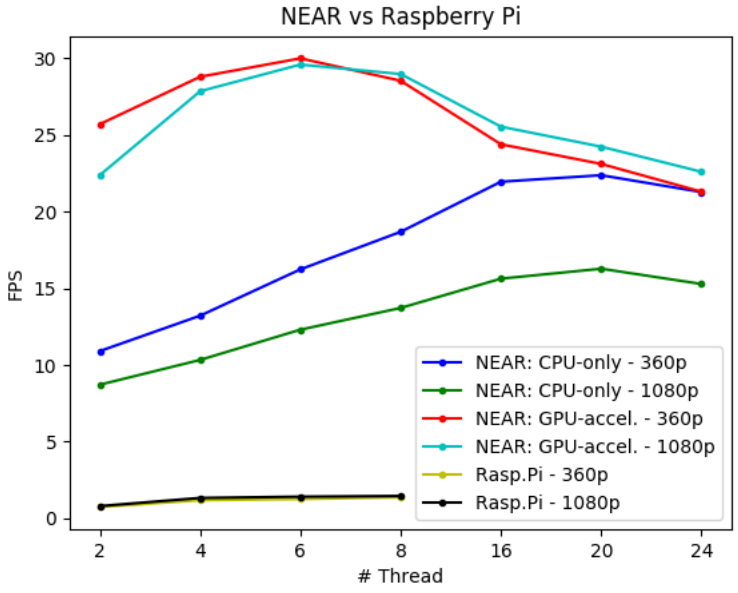


Figure 5.3: FPS comparison: NEAR vs Raspberry Pi

[2]To be clear, in GPU-accelerated mode, both CPU (running the specified amount of threads) and GPU are used.

Considerations

At first glance, NEAR better performs than Rasp. Pi according to a 10x factor in CPU-only mode and 15x in GPU-accelerated mode. Default values for both the pools of threads are the number of logical cores available in the system, namely 4 for *Rasp. Pi* and 40 for NEAR. However, NEAR-side, fig. 5.3 shows highest FPSs when using twenty threads (in both pools). This can be explained if we consider that NEAR's proxy server runs on a NUMA system and, when using more than 20 threads, 2 CPUs are used. In this way, the *cache-coherence* algorithm that takes place can lower the overall system performance.

RaspberryPi-side, maybe due to the low number of cores the system is equipped, although FPS are doubled when using 2 threads, they stay almost constant with the increasing of the number of threads.

In GPU-accelerated mode, the FPS peak is shifted when using less threads. Specifically, the highest FPS are reached when using six threads, compared to the CPU-only where they are achieved using twenty threads. In fact, GPU helps in terms of performance up to a certain value of CPU threads, namely six. After that, FPS gradually decrease until performance are worse than using CPU-only mode. To understand this behaviour, we have first counted the number of operations assigned to CPU and GPU and later noticed that when using less CPU threads, TensorFlow assigns more operations to GPU rather than to CPU. Moreover, TensorFlow, when needed, copies data from CPU to GPU to try to increase performance. However, when using a high number of threads, the number of copies from CPU to GPU linearly increases, resulting in worsening performance due to copy-overhead.

5.3.2 Latency comparison

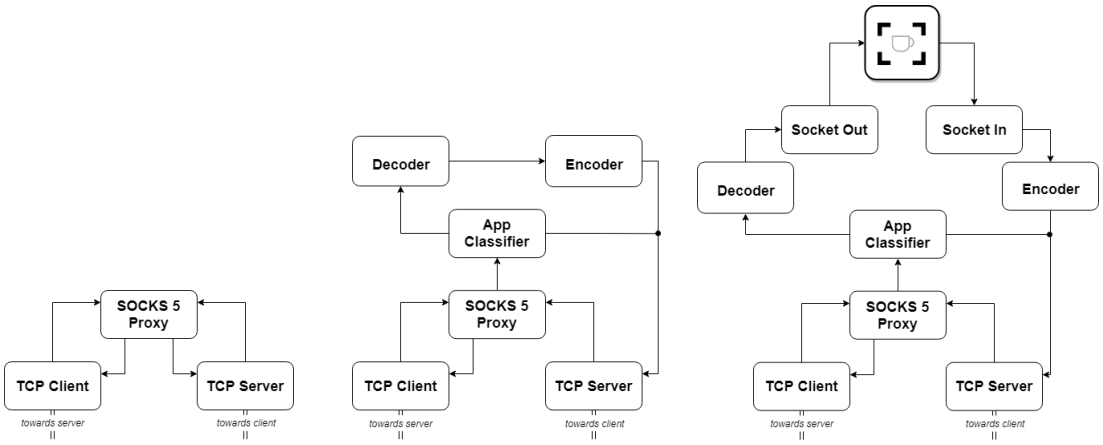
If on one hand offloading computing capabilities to a third party dramatically increases performances, on the other hand latency might be affected. In an on-device approach, latency can be considered to be zero, since data is retrieved where it has been generated. Instead, although taking advantage of (v)MEC to dramatically reduce latency overhead, NEAR introduces non-zero delay since data must be sent trough network anyway.

To quantify such delay, we considered a scenario where a client retrieves an on-line[3] streaming content from the Raspberry Pi. Client starts a vlc application to connect to server at time t_0. Server receives the request and starts content streaming. After ten seconds, server stops media streaming and closes connection. Client receives the last packet and automatically closes vlc at time t_1. Hence, the computed playout time with proxying is $P_p = t_1 - t_0$. Moreover, we launched the same experiment without using proxy in order to compute a direct playout time that we experienced being $P_d = 10.19$ seconds. Thus, we subtracted the latter to the previously computed playout time with proxying P_p to retrieve the playout delay $D = P_p - P_d$, shown in table 5.2.

To evaluate the overhead given by each NEAR component, we have considered three different scenarios in which media streaming is achieved using either only a proxy server

[3]Differently from the previous scenario where media was read by file, in this experiment the Pi Camera is used for on-line streaming.

fig. 5.4a, or proxy including re-encoding fig. 5.4b or the complete implementation with Object Detection fig. 5.4c, using different values of *step* (cf. section 4.4.5) in CPU-only or GPU-accelerated mode.



(a) with Proxy (b) with re-encoding (c) with Object Detection

Figure 5.4: Configurations for latency comparison

Proxy	**Re-encoding**	**Object Detection**		
			CPU-only	**GPU-accel.**
0.0015	1.32	**Step=10**	6.21	5.96
		Step=1	21.1	18.6

Table 5.2: Latency comparison

Considerations

Considering the experiment previously described, table 5.2 shows the retrieved results in terms of *playout delay (D)*, in seconds. First, it is important to note that proxying adds negligible delay, in the order of some milliseconds, compared to a end-to-end client-server connection. Second, stream re-encoding slightly increases delay adding a little over one second. This is still a good result if we consider a non-real time task.

However, the real bottleneck is given when performing Object Detection (OD). In CPU mode, when using a step value of *10* (i.e., OD is run once out of ten frames) video is still playable despite a delay of about *6* seconds. But, constantly applying Object Detection for each frame, meaning using a step equal to *1*, does make streaming unplayable with a dramatical increased delay up to *21* seconds[4].

Furthermore, if on one hand setting a step value of *10* makes CPU and GPU modes comparable, there is a visible difference when using a step value equal to *1* (i.e., OD

[4]The resulting steaming video is really unplayable since VLC media player skips frames that are too late.

is performed for each frame). This implies that GPU-accelerated mode gives substantial improvements if OD is widely exploited, meaning with a low step value. Thus, the fact that latency playout has comparable values with "high step" is reasonable since in this scenario the overhead is due to the used framework[5] rather than OD. To sum up, GPU does not help since using the lowest step value is not feasible with the implemented ClickNF framework.

This last result suggested to plot the evolution of the *playout delay (D)* when increasing the number of *step* in GPU-accelerated mode. The graph, shown in fig. 5.5, is composed by a first part where delay noticeably decreases until a step value of $S_x = 4$. After that, increasing the number of steps does not imply a substantial drop of delay. This means that S_x is a good trade-off value for achieving good performances and output quality[6].

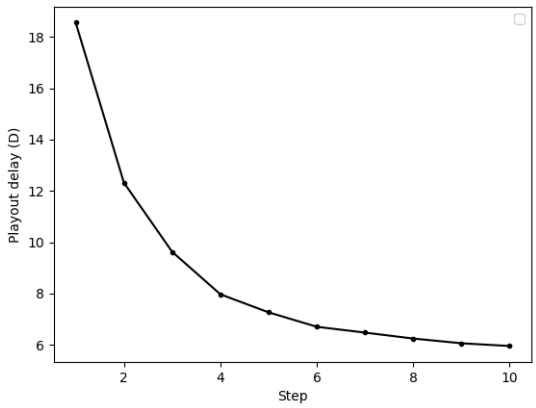


Figure 5.5: Playout delay evolution (GPU-accel. mode)

[5]ClickNF, and its elements, mainly Encoder and Decoder.

[6]Output quality in terms of refreshing frame statistics with updated information (cf. section 4.4.5).

Chapter 6

Conclusions

This final chapter finalizes the thesis work, summarizing the goals achieved and presenting some possible future works. As main achievement, we designed and implemented a fully-working prototype, namely NEAR, to be used for enabling legacy media devices, such as IP cameras, IoT devices, speakers and microphones, to transparently perform AR/VR tasks. NEAR consist on a proxy server running a NFV framework, namely ClickNF, and an AR task, Object Detection. The first provides flexibility and scalability to the whole architecture. For instance, NEAR is transparent to the endpoints (client and server) since they do not need special setup or hardware components. Nevertheless, clients can still request the legacy service without passing through NEAR, directly communicating with the legacy device.

To meet the ultra-low latency and high-computing power constraints required by those services, NEAR must be deployed at the edge infrastructure, in a network located near server and client, and should run on scalable machines (multi-core, multi-CPUs, ...) in order to perform the AR/VR task with satisfying quality results. To meet those requirements vMEC technology has been used. Specifically, we implemented NEAR on top of ClickNF, an existing NFV framework.

The well functioning of NEAR is also highlighted by the satisfying achieved results. In terms of performance (i.e., media output quality), NEAR has shown to be up to fifteen times better than using an on-device approach. As regards to latency, already aware to obtain by design slightly worse results compared to traditional implementations, and despite the need to implement framework optimizations, NEAR actually well performed also on this scenario resulting in adding an overhead of some unit of seconds in the worst case.

Although most of time has been spent in the implementation part of the project, NEAR is a prototype and there are still some useful features to add. First, NEAR has been tested with legacy codecs as MPEG1 and FLV, running on not-streaming-oriented protocols such as HTTP. This decision was necessary since, at the very beginning of the project, those were the only compatible with the set of framework used namely FFmpeg, Gstreamer (another set of media libraries) and OpenCV. In fact, we wanted to directly exploit FFmpeg and GStreamer[5] to avoid to "manually" perform stream re-encoding. However, due to unfeasibility of this approach, we had to change design keeping legacy technologies for convenience. Thus, modern codecs having higher compression factor

(such as H.265) can be tested and, eventually adapted to NEAR to work with.

Furthermore, NEAR currently supports playout of a streaming media from *one* server to *one* client. It means that our prototype cannot manage a scenario where multiple clients request one ore more streaming media to one or more servers. However, during implementation this aspect has always been kept into account, in order to ease the development of a multi-stream architecture in the future.

Finally, the ClickNF environment should be optimized, to allow speeding up performances in GPU-accelerated mode, and better exploited, implementing cross-layer optimizations. For instance, content caching with duplicate request suppression can be implemented to lower data overhead when multiple clients request the same stream. Moreover, client could be slightly modified to run remote commands (e.g., using REST APIs) that would allow to control (part of) the ClickNF graph at run-time. This would further increase client-side flexibility, for instance, allowing client to dynamically choose the AR/VR task to perform, or to setup media-related parameters (e.g., quality) at runtime.

Despite NEAR can be still enhanced, it is a valid and practical solution that allows common mobile and IoT devices (IP cameras, smartphones, microphones, speakers, ...) to perform AR/VR asks in a transparent, flexible and lightweight way.

Appendix A

Algorithms

A.1 App Classifier algorithm

Input: Packet p1
if *p1 contains SOCKS_APP_FLAG_ANNO* ***or*** *p1 contains TCP_DEL_FLAG_ANNO* **then**
  send p1 to port 1;
end
else
  send p1 to port 0;
end

Algorithm 1: App Classifier algorithm

A.2 Decoder algorithm

```
Input: Packet p
if D:status = HTTP_PARSE then
    D:http_header ← GET_HTTP_HEADER(p);
    D:status ← INITIALIZE;
    D:fd ← GET_FILE_DESCRIPTOR(p);
end
else
    if packet contains TCP_DEL_FLAG_ANNO then
        while queue.size > 0 do
            decode();
        end
        D:status ← LAST;
        D:nframe ← lastframe;
    end
    else
        put p in queue;
        while queue.size > QUEUE_SIZE do
            if D:status = INITIALIZE then
                initialize Decoder ;
                D:status ← DECODING;
            end
            decode();
        end
    end
end
```

Algorithm 2: *Decoder* algorithm

A.3 Object Detection algorithm

Input: String graphPath, String labelsPath, String sockIn, String sockOut, Int nthreads, Int threshold, Int step

```
— Main thread —
Create socket bound to path sockIn;
Connect to sockOut;
foreach client do
    Run Receiving thread;
    Run AR thread;
    Add Receiving thread to threadpool;
    Add AR thread to threadpool;
end
foreach thread t in threadpool do
    join t;
end

— Receiving thread —
while client is connected do
    Read frame from sockIn;
    lock_guard(mutex);
    Push frame into queue;
    condv.notify_one();
end

— AR thread —
Load graph from graphPath;
Load labels from labelsPath;
while true do
    condv.wait(uniquelock(mutex), !queue.empty());
    Pop frame from queue;
    lock.unlock();
    if frame is null then
        break;
    end
    frame ← ObjectDetection(frame);
    Send frame to sockOut;
end
```

Algorithm 3: *Object Detection* algorithm

A.4 Encoder algorithm

```
Input: Packet p
if status = HTTP_SEND then
    send D:http_header to port 0;
    status ← INITIALIZE;
end
else
    buffer[pos] = p.data;
    pos ← pos + p.len;
    if pos = size then
        if status = INITIALIZE then
            initialize Encoder ;
            status ← ENCODING;
        end
        encode();
        if D:status = LAST and D:nframe = lastframe then
            send FIN;
        end
        pos ← 0;
    end
end
```

Algorithm 4: *Encoder* algorithm

Bibliography

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[2] J. W. Anderson et al. "xOMB: Extensible Open MiddleBoxes with commodity servers". In: *2012 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2012, pp. 49–60.

[3] Hitesh Ballani et al. "Enabling End-Host Network Functions". In: *SIGCOMM Comput. Commun. Rev.* 45.4 (Aug. 2015), pp. 493–507. ISSN: 0146-4833. DOI: 10.1145/2829988.2787493. URL: http://doi.acm.org/10.1145/2829988.2787493.

[4] Tom Barbette, Cyril Soldani, and Laurent Mathy. "Fast Userspace Packet Processing". In: *Proceedings of ANCS 2015*. 2015.

[5] Fabrice Bellard. *FFmpeg*. https://www.ffmpeg.org/. [Online; accessed 11-January-2019].

[6] Anat Bremler-Barr, Yotam Harchol, and David Hay. "OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: ACM, 2016, pp. 511–524. ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2934875. URL: http://doi.acm.org/10.1145/2934872.2934875.

[7] Anat Bremler-Barr et al. "Deep Packet Inspection as a Service". In: *CoNEXT*. 2014.

[8] Tiffany Yu-Han Chen et al. "Glimpse: Continuous, real-time object recognition on mobile devices". In: *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. ACM. 2015, pp. 155–168.

[9] Ronan Collobert, Samy Bengio, and Johnny Marithoz. *Torch: A Modular Machine Learning Software Library*. 2002.

[10] Michael Copeland. *What's the Difference Between Deep Learning Training and Inference?* Tech. rep. NVIDIA, 2016.

[11] I. Culjak et al. "A brief introduction to OpenCV". In: *2012 Proceedings of the 35th International Convention MIPRO*. 2012, pp. 1725–1730.

[12] Xiaoyan Pei et al. “Network Functions Virtualisation”. In: *SDN and OpenFlow World Congress* (2012).

[13] ETSI. *Multi-access Edge Computing*. https://www.etsi.org/technologies-clusters/technologies/multi-access-edge-computing. [Online; accessed 02-January-2019].

[14] OpenCV Foundation. *OpenCV*. https://opencv.org/. [Online; accessed 11-January-2019].

[15] The Linux Foundation. *Data Plane Development Kit*. https://www.dpdk.org/. [Online; accessed 02-January-2019].

[16] GitHub. *Tensorflow detection model zoo*. https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md. [Online; accessed 12-January-2019].

[17] Google. *Bazel*. https://bazel.build/. [Online; accessed 11-January-2019].

[18] Google. *Google Assistant*. https://assistant.google.com/. [Online; accessed 02-February-2019].

[19] Google. *Google Cloud*. https://cloud.google.com. [Online; accessed 02-February-2019].

[20] Google. *TensorFlow's Object Detection*. https://github.com/tensorflow/models/tree/master/research/object_detection. [Online; accessed 11-January-2019].

[21] *GSteamer*. https://gstreamer.freedesktop.org/. [Online; accessed 02-February-2019].

[22] Remzi H. and Andrea C. *Operating Systems*. Arpaci-Dusseau Books. Chapter: Condition Variables. 2014.

[23] Kiryong Ha et al. “Towards wearable cognitive assistance”. In: *Proc. of the 12th annual international conference on Mobile systems, applications, and services. ACM*. 2014.

[24] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. “NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms”. In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 445–458. ISBN: 978-1-931971-09-6. URL: http://dl.acm.org/citation.cfm?id=2616448.2616490.

[25] Intel. “An Introduction to the Intel® QuickPath Interconnect”. In: *Intel* (2009).

[26] Intel. *Nervana (Neural Network Processor)*. https://www.intel.ai/. 2017.

[27] Internet Engineering Task Force. *RFC 1928 - SOCKS Protocol Version 5*. https://www.ietf.org/rfc/rfc1928.txt. 1996.

[28] Eun Young Jeong et al. “mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems”. In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 489–502. ISBN: 978-1-931971-09-6. URL: http://dl.acm.org/citation.cfm?id=2616448.2616493.

[29] Yangqing Jia et al. “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *Proceedings of the 22Nd ACM International Conference on Multimedia*. MM ’14. Orlando, Florida, USA: ACM, 2014, pp. 675–678. ISBN: 978-1-4503-3063-3. DOI: 10.1145/2647868.2654889. URL: http://doi.acm.org/10.1145/2647868.2654889.

[30] David Patterson Kaz Sato Cliff Young. “An in-depth look at Google’s first Tensor Processing Unit (TPU)”. In: *Google Cloud Big Data and Machine Learning Blog* (2017).

[31] Joongi Kim et al. “A high-performance packet processing framework for heterogeneous processors”. In: *Proceedings of the Tenth European Conference on Computer Systems. ACM, 22*. 2015.

[32] E. Kohler. *Click - GitHub*. https://github.com/kohler/click. [Online; accessed 02-January-2019].

[33] Eddie Kohler et al. “The Click Modular Router”. In: *ACM Transactions on Computer Systems (TOCS)* (2000).

[34] Nokia Bell Labs. *ClickNF - GitHub*. https://github.com/nokia/ClickNF. [Online; accessed 02-January-2019].

[35] Rafael Laufer et al. “CliMB: Enabling Network Function Composition with Click Middleboxes”. In: *ACM SIGCOMM Computer Communication Review* 46 (Dec. 2016), pp. 17–22. DOI: 10.1145/3027947.3027951.

[36] Guyue Liu et al. “Microboxes: high performance NFV with customizable, asynchronous TCP stacks and dynamic subscriptions”. In: Aug. 2018, pp. 504–517. DOI: 10.1145/3230543.3230563.

[37] Peng Liu, Bozhao Qi, and Suman Banerjee. “Edgeeye: An edge service framework for real-time intelligent video analytics”. In: *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking*. ACM. 2018, pp. 1–6.

[38] R. Laufer M. Gallo. “ClickNF: a Modular Stack for Custom Network Functions”. In: *USENIX* (2018).

[39] Duncan MacMichael. “Introduction to Receive Side Scaling”. In: *Microsoft Hardware Dev Center* (2017).

[40] Ilias Marinos, Robert N. M. Watson, and Mark Handley. “Network Stack Specialization for Performance”. In: vol. 44. Nov. 2013. DOI: 10.1145/2535771.2535779.

[41] Joao Martins et al. “ClickOS and the Art of Network Function Virtualization”. In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI’14. Seattle, WA: USENIX Association, 2014, pp. 459–473. ISBN: 978-1-931971-09-6. URL: http://dl.acm.org/citation.cfm?id=2616448.2616491.

[42] Anandatirtha Nandugudi et al. “Network Function Virtualization: Through the Looking-Glass”. In: *Springer Annals of Telecommunications manuscript* (2016).

[43] Nokia. *Virtualized MEC*. https://networks.nokia.com/solutions/multi-access-edge-computing. [Online; accessed 02-January-2019]. 2019.

[44] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[45] Narasimha Prasanna. *A beginner introduction to TensorFlow (Part-1).*

[46] *Raspberry Pi.* https://www.raspberrypi.org/. [Online; accessed 11-January-2019].

[47] Redis. *TCP localhost vs Unix socket.* https://redis.io/topics/benchmarks. [Online; accessed 11-January-2019].

[48] Seyyed Salar Latifi Oskouei et al. "CNNdroid: GPU-Accelerated Execution of Trained Deep Convolutional Neural Networks on Android". In: Oct. 2016, pp. 1201–1205. DOI: 10.1145/2964284.2973801.

[49] Samsung. *Samsung Gear VR.* https://www.samsung.com/global/galaxy/gear-vr/. [Online; accessed 02-February-2019].

[50] Frank Seide and Amit Agarwal. "CNTK: Microsoft's Open-Source Deep-Learning Toolkit". In: *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* KDD '16. San Francisco, California, USA: ACM, 2016, pp. 2135–2135. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2945397. URL: http://doi.acm.org/10.1145/2939672.2945397.

[51] Vyas Sekar et al. "Design and Implementation of a Consolidated Middlebox Architecture". In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12).* San Jose, CA: USENIX, 2012, pp. 323–336. ISBN: 978-931971-92-8. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/sekar.

[52] R. Sjoberg et al. "Overview of HEVC High-Level Syntax and Reference Picture Management". In: *IEEE Transactions on Circuits and Systems for Video Technology* 22.12 (2012), pp. 1858–1870. ISSN: 1051-8215. DOI: 10.1109/TCSVT.2012.2223052.

[53] Radu Stoenescu et al. "In-Net: in-network processing for the masses". In: *EuroSys.* 2015.

[54] Marco Trinelli, Massimo Gallo, and Fabio Pianese. "NFV for artificial intelligence". In: *SDN Day 18.* 2018.

[55] Marco Trinelli et al. "Transparent AR Processing Acceleration at the Edge". In: *EdgeSys* (2019).

[56] VideoLAN. *VLC Media Player.* https://www.videolan.org/vlc. [Online; accessed 03-February-2019].

[57] *WebRTC.* https://webrtc.org/.

[58] Wikipedia. *Augmented reality — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Augmented%20reality&oldid=875790325. [Online; accessed 02-January-2019]. 2019.

[59] Wikipedia. *Cloud computing — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Cloud%20computing&oldid=871497546. [Online; accessed 02-January-2019]. 2019.

[60] Wikipedia. *Kinect — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Kinect&oldid=874442295. [Online; accessed 02-January-2019]. 2019.

[61] Wikipedia. *OpenCV — Wikipedia, The Free Encyclopedia*. http://it.wikipedia.org/w/index.php?title=OpenCV&oldid=99378982. [Online; accessed 03-February-2019]. 2019.

[62] Wikipedia. *PlayStation VR — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=PlayStation%20VR&oldid=872908552. [Online; accessed 02-January-2019]. 2019.

[63] Wikipedia. *Pokémon Go — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Pok%C3%A9mon%20Go&oldid=876331871. [Online; accessed 02-January-2019]. 2019.

[64] Wikipedia. *Virtual reality — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Virtual%20reality&oldid=875679564. [Online; accessed 02-January-2019]. 2019.

[65] Google X. *Google Glass*. https://www.x.company/glass/. [Online; accessed 02-February-2019].

[66] Wei Zhang et al. "SDNFV: Flexible and Dynamic Software Defined Control of an Application- and Flow-Aware Data Plane". In: *Proceedings of the 17th International Middleware Conference*. Middleware '16. Trento, Italy: ACM, 2016, 2:1–2:12. ISBN: 978-1-4503-4300-8. DOI: 10.1145/2988336.2988338. URL: http://doi.acm.org/10.1145/2988336.2988338.

Glossary

Access Control List (ACL) is a list of permissions, linked to an object (file), that specifies which users have access to that object. The access can be usually read, write, or both.

Application Programming Interface (API) is a set of methods, protocols and libraries that makes it easier to develop a computer program by providing high-level building blocks, which are then combined together to achieve the desired function.

Artificial Neural Networks (ANN) are inspired by the biological neural networks with the function to learn tasks by considering real-life examples. An ANN is based on a collection of connected nodes called artificial neurons. Each connection, like the synapses in a biological brain, can transmit a signal from one artificial neuron to another.

Artificial Intelligence (AI) is intelligence demonstrated by machines. It is seen as an imitation of the human intelligence that learning from practice solves new and more complex problems. It usually deals with analyzing big quantities of data in a small amount of time.

Augmented Reality (AR) is a computer-generated interactive experience where common objects are displayed with additional useful information. Usually no specific hardware is needed.

Capital Expenditure (CaPex) is the cost to develop or buy long terms assets for a product or system.

Cloud Computing is a model for enabling convenient and on-demand services using a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort.

Codec is a software/hardware utility or file rapresentation used to compress (at source) and decompress (at destination) media content in order to minimize system storage and network bandwidth.

Direct Memory Access (DMA) is a mechenamism that allows compatible peripherals to directly access the internal memory to read/write blocks of data, instead

of using traditionals CPU's interrupts for each data request, resulting in an overall increase of performances.

Internet of Things (IoT) is a set of network devices able to interact and exchange data each others using the Internet. Common scenarios are vehicles and home appliences.

Machine Learning (ML) is an application of Artificial Intelligence (AI) that exploits statistical models to provide systems the ability to automatically learn and improve from experience without being explicitly programmed.

Multi-access Edge Computing (MEC) sometimes shortened "Edge Computing", is a network architecture framework that enables Cloud Computing facilities and an IT services at the edge of a (cellular) network.

Network Function Virtualization (NFV) is a network framework concept that uses the technologies of IT virtualization to *softwarize* hardware devices deploying network functions in order to create building blocks that are usually connected together.

Non-Uniform Memory Access (NUMA) is a memory architecture defined for multiprocessor systems where *memory access time* depends on the logical position of the processor. In NUMA architectures, a processor can access its own local memory rapidly than shared memories (local to another processors).

Operating Expenditure (OpPex) is the cost needed to the managment of a product, service or system.

Resource Acquisition Is Initialization (RAII) is a widely known paradigm used in object-oriented programming. Objects exploiting RAII, have a constructor that internaly allocates/acquires a resource that is, at the end, automatically deallocated/released in the destructor. It is used to prevent issues concerning memory leakage and race conditions.

Streaming media is a multimedia content received and played by an endpoint (client) while being delivered by a provider (server). The term is in contrast with *media downloading*, where the content is played by the user only after having received the whole media file.

Transcoding is the process of converting the codec (and thus the format) of a media content. The destination codec can also be the same (with eventual different media-related options).

Virtual Reality (VR) like AR, is an interactive experience. The difference with AR is that VR creates a full-fledged virtual envirorment where the user, impersonating an avatar, can enjoy a dynamic experience (walking, shooting, playing tennis, ...).