University of Zurich^UZH

GENEVIZ — Generation, Validation, and Visualization of SFC Packages

Martin Juan José Bucher
Zurich, Switzerland
Student ID: 15-705-015

Supervisors: Muriel Figueredo Franco, Eder John Scheid
Date of Submission: May 13, 2019

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Zusammenfassung

Network Function Virtualization (NVF) verfolgt das Ziel, die Paketverarbeitung von Netzwerkfunktionen durch den Einsatz von Virtualized Network Functions (VNFs) auf generischer Standard-Hardware von dezidierten Hardwaregeräten zu entkoppeln. Netzbetreiber können massgeschneiderte Netzwerkdienste erstellen, indem mehrere VNFs miteinander verkettet werden, was auch als Service Function Chaining (SFC) bezeichnet wird. Obwohl NFV immer populärer und technisch ausgereifter wird, verlangt die Konstruktion solcher SFCs noch immer fundierte Kenntnisse über die NFV-Technologie. Zudem kann die Erstellung eines SFC bisher nur manuell gemacht werden. In der vorliegenden Arbeit stellen wir GENEVIZ vor, ein Tool, welches ein *benutzerfreundliches Interface* sowohl für die Konstruktion und Generierung komplett neuer SFCs von Grund auf, als auch für den Import und die Anpassung bereits zuvor erstellter SFCs bereitstellt. Letzteres ist insbesondere deshalb interessant, weil somit neue SFCs erstellt werden können, welche auf zuvor schon existierenden SFCs basieren. Darüber hinaus gehen wir das Thema Datenintegrität an und bieten einen Weg, um SFCs, die aus einer externen Quelle bezogen wurden, durch die Verwendung der Blockchain-Technologie validieren zu können. GENEVIZ zielt darauf ab, die Erstellung von SFCs *intuitiver* und *einfacher* zu machen. Zugleich wird die Anzahl der Schritte, welche notwendig sind für die verschiedenen Anwendungsfälle, reduziert. Wir führen drei Fallstudien an unserem entwickelten Prototypen durch und zeigen dadurch nicht nur die technische Machbarkeit von GENEVIZ, sondern weisen auch die Nutzbarkeit der verschiedenen Visualisierungen nach.

Abstract

Network Function Virtualization (NFV) aims to decouple the package processing of network functions from dedicated hardware appliance by running Virtualized Network Functions (VNFs) on general-purpose hardware. Network operators can create customized network services by chaining multiple VNFs together, forming a so-called Service Function Chaining (SFC). Although NFV becomes more popular and technically mature, the construction of such SFCs still needs in-depth knowledge about NFV technology. Furthermore, the creation of an SFC can only be done manually up until now. In this thesis, we introduce GENEVIZ, a tool providing a *user-friendly interface* both for the construction and generation of completely new SFCs from scratch as well as for the import and adjustment of previously created SFCs in order to create new SFCs based on existing ones. Beyond that, we address the issue of data integrity and give the possibility to validate SFCs — received from an external source — through the usage of blockchain technology. GENEVIZ aims to provide a way to create SFCs *more intuitive* and *easier*. In addition, the number of steps necessary for different use cases is reduced. We conduct three case studies on our developed prototype, not only showing the technical feasibility of GENEVIZ, but also providing evidence of the usability of the different visualizations we proposed.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Muriel Figueredo Franco, for his continuous assistance and inputs during the writing of this thesis. It has been a tremendous pleasure to work with someone as passionate and smart as Muriel, as I could not only profit from his insights for the draw up of this thesis but also on how to think from a scientific perspective in the research area of *Computer Science*.

I would also like to thank my co-supervisor, Eder John Scheid, for his precious inputs on GENEVIZ from a more abstract layer, and especially his help during the blockchain implementation for the Prototype.

Finally, I would like to thank Prof. Dr. Burkhard Stiller, head of the Communication Systems Research Group (CSG) at the University of Zurich, for making it possible to write my bachelor's thesis within this hot research topic.

Contents

Chapter 1

Introduction

The paradigm of Network Function Virtualization has gathered significant attention over the last couple of years both from academia as well as from industry, leading to a disruptive shift in telecommunication service provisioning [1]. NFVs decouple the packet processing from dedicated hardware middleboxes and handle it on Virtual Network Functions instead, running on commercial off-the-shelf programmable hardware, such as general-purpose servers, storage and switches [2]. Besides the technical challenges of NFV, as the paradigm is still in its infancy, it comes with several benefits, including simplified network operations, a potential of speeding up service delivery, and significant reductions in Operational Expenditures (OPEX) and Capital Expenditures (CAPEX) [3]. With the usage of NFV, network operators can create customized network services by chaining together multiple VNFs. Such an aggregation of different network functions builds up a network service, fulfilling the individual demand of the user.

1.1 Motivation

Nowadays, a network operator should have in-depth knowledge about NFV technology in general and its corresponding descriptors in order to create a Service Function Chaining, which it can be deployed on an NFV-enabled infrastructure in the end. In NFV, an SFC is represented as a forwarding graph of VNFs and should take into account three main descriptors:

(i) **VNF Descriptor (VNFD)**: Represents a VNF by defining deployment and operational behavior requirements. A single SFC usually contains several of those.

(ii) **VNF Forwarding Graph Descriptor (VNFFGD)**: Provides a textual representation of the VNF Forwarding Graph (VNFFG), information about the VNFDs which compound the service, virtual links, and dependencies between particular VNFs.

(iii) **Network Service Descriptor (NSD)**: Contains information about the network service vendor and specific requirements used for the deployment of the service on the network.

We argue that the process of constructing such an SFC is not very intuitive, a lot of manual steps are necessary for the file handling and editing, and the creation can be — for inexperienced users — quite error-prone. This might even lead to a negative impact on the broad adoption of NFV technology in general. Furthermore, by considering the prospective market growth of *VNF as a Service* (VNFaaS) [4] and its potential to simplify the way how end users obtain VNFs and services in general, the lack of intuitive solutions should also be addressed when considering the potential of SFCs for end users without a lot of expertise in NFV technology.

In this context, Information Visualization techniques can be a powerful tool, helping network administrators understand the behavior of the managed network or service in a manner which is quicker and easier [5]. Even though the analysis of such data can be almost fully automated, human interpretation plays a crucial role in the interpretation and decision-making while doing network and service management. Especially in NFV environments, there is an enormous amount of data available and the understanding and evaluation of it represent a challenging task itself [6].

Although past work explored visualization techniques to simplify the identification of problems in SFCs, there is still a lack of research addressing the simplification of SFC construction. We claim that information visualization could provide several benefits in NFV environments. Such benefits include (i) an intuitive way to select VNFs that will compound the forwarding graph, (ii) a quick configuration of the SFC through its corresponding VNFs in order to meet particular requirements, (iii) a fast identification of SFC misconfigurations, and (iv) the validation of the data integrity of a previously created SFC. Besides all those, a visual interface can also provide opportunities to reuse already available SFCs as a template and configure them to meet new requirements, making it much easier to build a new SFC based on an existing one.

1.2 Description of Work

The present bachelor thesis covers first the design and implementation of a visualization tool, which allows the construction of a new SFC Package based on multiple VNF Packages and other inputs defined through the visualization tool. The interface should be as intuitive as possible, providing an easy way to create new SFCs as well as to configure properties of the VNFs the SFC consists of. Secondly, the provided tool should be able to store the hash of the content of such a package in a public blockchain. Thus, network operators can validate an already existing SFC configuration before it gets deployed inside their network. Finally, an evaluation of the usability of the provided solution should be conducted.

1.3 Thesis Outline

The remainder of this thesis is structured as follows: Chapter 2 gives a rough overview of several concepts of utmost significance for the context of this thesis. In Chapter 3, the design and general architecture of the visualization tool are discussed, whose implementation is addressed in Chapter 4. An evaluation of the effectiveness of the prototype which was built is conducted in Chapter 5. Chapter 6 identifies key research areas for possible future exploration, followed by Chapter 7 concluding the present work.

Chapter 2

Background and Related Work

The following sections aim to give a preliminary overview of the major technological concepts that are building the fundamental basis for the present bachelor thesis. First, we will discuss a few important architectural concepts leading the current shift in computer networking, followed by a rough summary of the emerging blockchain technology and information visualization since they all fit into the context of this thesis.

2.1 Network Function Virtualization

Traditionally, network operators are confronted with an extensive variety of hardware appliances. For each function of a given service one needs to deploy dedicated physical devices and equipment with the service components having a strict ordering inside a network topology. These specific spatial placements — together with the demand for high stability, quality, and strict protocol compliance — have led to heavy dependence on specialized hardware, long product cycles, and low service agility [1]. NFV aims to resolve those issues by virtualizing industry network equipment onto industry standard general-purpose hardware.
One of the most crucial works was published in 2012 by the Industry Specification Group (ISG) under the auspices of the European Telecommunications Standards Institute (ETSI) [7]. Therein, they summarized the benefits and challenges involved with an architecture supporting Network Function Virtualization with the goal to encourage international collaboration as well as to accelerate the development of the new paradigm. A large amount of work has been published since then, exploring possible solutions for single parts of its architecture and the NFV system as a whole. Among various advantages coming with NFV — with the potential of leading a radical shift in the telecommunications industry — we can summarize the main benefits as follows:

1. **Efficiency** — By consolidating multiple network functions on one single hardware and exploiting the economies of scale of the IT industry, both the amount of capital investment as well the energy consumption can be reduced in order to save costs. Dedicated hardware for specific networking functions would be omitted in the near

future, focusing research to a reduced variety of hardware equipment. The base of know-how for general-purpose computers is much larger than the one for the specific equipment in the telecommunication industry.

2. **Faster Time to Market** — Through NFV, the time needed to deliver new services on the market is drastically reduced. The typical innovation cycle of network operators can be minimized by deploying new software completely remotely instead of going there physically to install new software. Further, the amortization costs for the infrastructure are reduced by the effects of economies of scale.

3. **Flexibility** — The support of multi-tenancy allows network operators to provide targeted and tailored services based on the customer's needs, as the NFV infrastructure can easily scale up or out resources for certain services. Even though that would be possible on a traditional networking system as well, NFV can scale much faster, making the system more agile for abrupt changes in traffic or demand. Also, the mitigation of failures within a network could be done automatically by the reconfiguration and bypassing of traffic through orchestration mechanisms.

With all the advantages coming along with NFV there are also some technical challenges which need to be addressed in order to enable fast and broad adoption of NFV [8]:

1. **Performance and Stability** — Latency and throughput should remain stable over time. Previous work has shown significant variations in both of them, which should not be the case in a productive system in the near future. There is also a processing overhead for the virtualization which could affect performance as well. The key challenge is to keep a networking system with a virtualization layer at least as good as a system consisting of physical dedicated appliances.

2. **Migration** — The effort for a smooth transition from existing networks to NFV-enabled systems will confront network providers with big challenges when considering the massive scale of today's globally spanning network and the tight coupling among components. In theory, a hybrid version could be composed from physical and virtual network appliances with the final goal to replace the physical ones step by step.

3. **Placement of VNFs** — The separation of functionality and physical location brings up the question of how to efficiently place the virtual functions inside a network. Ideally, network operators would place a VNF where it will be least expensive and used most effectively. As network functions often need to fulfill certain requirements regarding latency and throughput, the placement of a VNF will be of significance for network operators.

2.1.1 NFV Architecture

Virtualization allows a very flexible way to run software in general. Regarding NFV, it enables dynamic schemes for the creation and management of network functions [9] due to three fundamental differences: (i) the software is separated from the hardware which allows independent evolvements of hardware and software respectively, (ii) services can be provided in a dynamic way by scaling up or down resources as there is a need for it, and (iii) network functions can be deployed very flexible on a pool of hardware resources running at different times in different locations. A high-level architecture of the NFV framework is illustrated in Figure 2.1 and consists of five major functional blocks:

(i) **Orchestrator** — Responsible for the orchestration and management of software resources and the virtualized infrastructure of the hardware for the realization of networking services.

(ii) **VNF Manager (VNFM)** — Communicates with the respective VNFs in order to instantiate, terminate, scale or update VNFs during their life cycle. Also, information about configurations and events (e.g. warnings or errors) are stored by the VNFM. This block could be automized almost completely in the future, solving issues with productive VNF instances by itself.

(iii) **NFV Infrastructure (NFVI)** — The virtualization layer in this block abstracts the physical hardware from the virtualized components the VNFs are addressing to. By providing standardized interfaces, the life cycle of a VNF remains independent from the hardware underneath.

(iv) **Virtualized Infrastructure Manager (VIM)** — Responsible for the virtualization and management of the given computing, networking and storage resources provided by the real hardware. A ”parent” NFVI may contain more than one VIM, with each VIM responsible for one single NVFI from a given infrastructure.

(v) **Virtualized Network Functions (VNFs)** — Moves the network functions from dedicated appliances to software-based functions. A typical VNF, for example, is a Firewall, Content Delivery Network (CDN), or Network Address Translation (NAT).

The three components VNFM, VIM, and Orchestrator together form the so-called *NFV Management and Orchestration (MANO)* sub-framework which is of crucial importance for the understanding and management of an NFV environment. The MANO is illustrated as the dashed box on the right side in Figure 2.1.

2.2 Service Function Chaining

The chaining of service functions can be described as the mechanism of defining an ordered list of network functions and stitching them together to a single chain. This virtual chain can be specified as a **Service Function Chaining (SFC)**. Thus, an SFC can be seen as a graph describing a set of Network Functions (NF), connection points, as well as the order in which the packets are traversing the NFs in the given path [10].
Such a network function doesn’t necessarily rely exclusively on VNFs and can also run on commercial physical hardware appliances. In general, an SFC is defined and configured by a network operator which has expertise in the field of the operating network functions

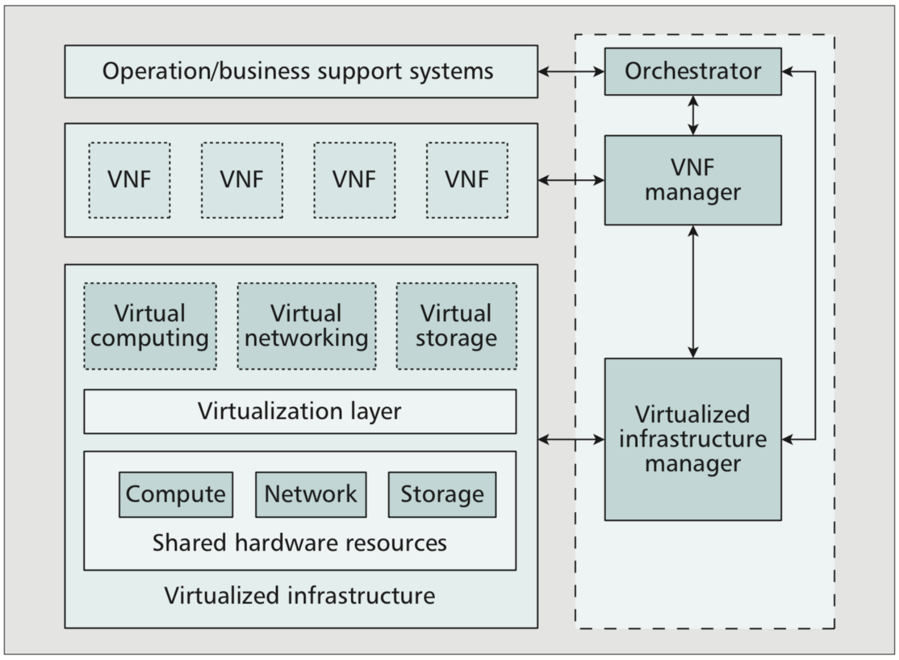


Figure 2.1: NFV Architecture [3]

working underneath. The paths along with the network often need to satisfy certain requirements in terms of bandwidth efficiency. The network operator, therefore, has to provide for these demands.

It's important at this point to denote that NFV is closely related to **Software Defined Networking (SDN)**. Although both are complementary and can benefit from each other, they do not depend on each other. SDN can be seen as a network paradigm characterized by the following three properties [11]:

(i) The data forwarding plane and control planes are clearly separated.

(ii) The network logic is completely abstracted from the hardware implementation into software.

(iii) There is a network controller coordinating the forwarding decisions of involved network devices.

NFV can be implemented without SDN and vice versa. However, the separation of the control and data forwarding plane, as suggested by SDN, can simplify the compatibility with existing deployments, enhance performance, and alleviate maintenance and operation procedures [12]. SDN can also enable dynamic and cost-efficient traffic steering among individual VNFs [13].

Coming back to SFC, we briefly want to discuss two uses cases given in Figure 2.2. Both scenarios aim to satisfy certain end user requirements, accomplished through the chaining of different NFs. The first use case addresses the delivery of a web service, the second use case a video service. Without SDN, the traffic forwarding between NFs needs to be manually configured by a network operator on certain forwarding devices (i.e. a router or a switch). Without NFV the scaling or update of certain NFs with increased traffic will get harder to manage the more NFs the SFC contains. Hence, the combination of SDN and NFV can facilitate a flexible and efficient SFC provisioning. Additionally, with the combination of SDN and NFV, SFCs could use network resources more efficiently, allowing certain VNFs in the network to adjust their CPU usage, allocated memory, or disk size for storage.

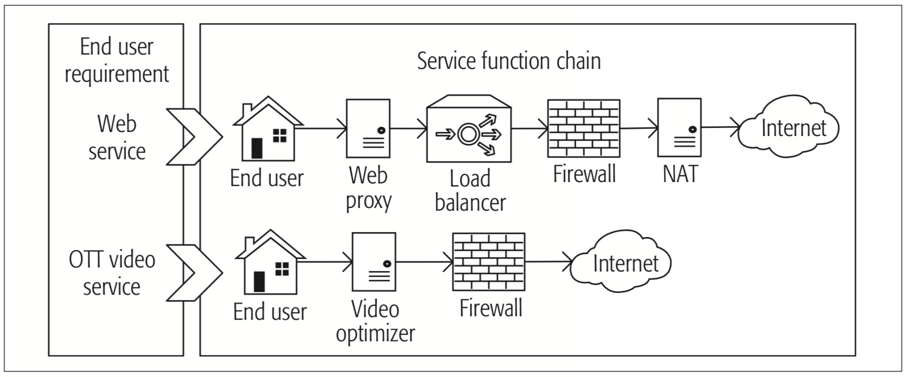


Figure 2.2: SFC structure by means of two use cases [10]

2.2.1 Network Function Descriptors

In the following section, we aim to discuss the three descriptors mentioned in the introduction in a more detailed manner. All three of them are — in theory — part of the composition of our proposed *SFC Package*, which aims to include all information necessary to deploy a new network service on an NFV environment:

VNF Descriptor (VNFD)

The Virtual Network Function Descriptor describes a VNF regarding its operational and deployment requirements. It is not only necessary for the usage by NFV MANO for the creation of a virtual link within an NFVI but also contains information about the behavior of the VNF over a period of time. Additionally, the VNFD also contains information about the management of its corresponding VNF, such as several parameters for monitoring the VNF as well as a description of the minimum resource requirements necessary to run the VNF properly. Of greater importance are the connection points defined in the VNFD, which are needed to establish appropriate Virtual Links within the NFVI between different VNF Components (VNFCs) [9].

Network Service Descriptor (NSD)

The Network Service Descriptor consists of static information used by the NFV Orchestrator (NFVO) for the instantiation of a new network service. It can contain information necessary to deploy VNFs as well as information about the network service vendor. The NSD can also include references to other nested NSDs, Virtual Link Descriptors (VLDs), Physical Network Function Descriptors (PNFDs), VNFDs and the VNF Forwarding Graph Descriptor (VNFFGD).

VNF Forwarding Graph Descriptor (VNFFGD)

The VNF Forwarding Graph Descriptor contains meta information about the VNF Forwarding Graph (VNFFG) itself as well as references to VNFs, PNFs, and VLs. It is still a challenge nowadays how the VNFFGD will track changes from the deployed instances (e.g. an additional VNF is connected for scaling up the service) as the forwarding graph and therefore the descriptor itself must change. Hence, similar to the way the VNFD describes a VNF, the VNFFGD describes a VNFFG.

2.3 Blockchain Technology

Although the underlying technologies for the blockchain were already developed decades ago, it has not gained any significant attention up until the publication of a whitepaper [14] written by Satoshi Nakamoto in 2008. In there, Nakamoto introduced *Bitcoin*, a digital currency providing a solution to the problem of trust in a decentralized system. The ledger of Bitcoin is publicly auditable for all participants (peers) in the network. Transactions are broadcasted to all peers and their validity is verified by each peer independently [15]. Multiple valid transactions are collected into a single block, which is then sealed cryptographically. A new block follows up on the predecessor by referencing it, hence forming a chain of blocks, and is generated by special peers (also called voters or miners) based on a cryptographic puzzle. Usually, each block contains the hash of the previous block, computed through a cryptographic hash function.

A cryptographic hash function is a special subtype of a hash function and has certain properties, making it especially suitable for cryptography. Typically, it has the following five properties:

(i) It is one-directional (one can't reverse it easily to find its input)
(ii) The computation of the hash is fast
(iii) Finding the same hash value for two different inputs is infeasible
(iv) It is deterministic in the sense that it will always produce the same output for the same input
(v) A small change in the input should result in a large change of the output, making it impossible to map two similar outputs to two similar inputs

As the hash of the previous block is contained in the header of the newly created block, any alteration of the previous block results in a different hash of the content. Thus, all transactions contained in a newly mined block are made **immutable**. Any modification of a transaction previously made would make the block and hence the whole blockchain *invalid*.

Bitcoin uses a *Merkle Tree* for the transactions, where every leaf node holds the hash of a certain data block, and every non-leaf node holds the cryptographic hash of the child nodes. Only the root is included in the block's hash in order to save storage as shown in Figure 2.3. But not all distributed ledgers consist of a chain of blocks such as Bitcoin does. Thus, a blockchain is just one type of distributed-ledger technology and there exist many other systems already. IOTA [16] for example takes usage of a *Tangle* — a new data structure based on a directed acyclic graph — instead of blocks.

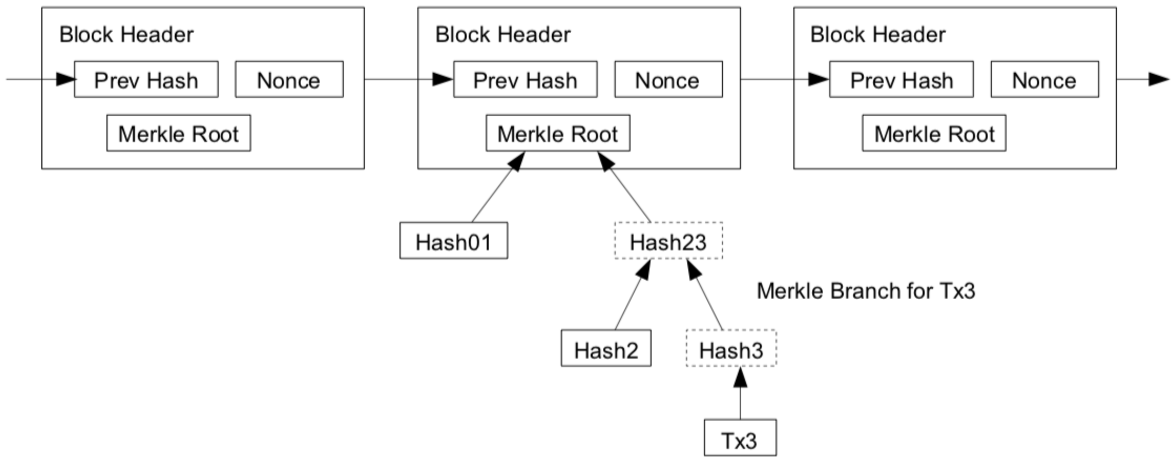


Figure 2.3: Blockchain concept as seen in the Bitcoin Whitepaper [14]

The competition among different miners of the Bitcoin Blockchain is based on their respective computational power. This kind of consensus mechanism is referred to as *Proof of Work*. Not all blockchain architectures rely on the *Proof of Work* consensus algorithm and there exist other consensus methods such as *Proof of Stake*, where a set of validators take turns proposing and voting on the next block, and the weight of the vote of each validator depends on the deposit's size. Another consensus mechanism is *Proof of Authority*, where a fixed set of block creators and validators is defined. But besides the mentioned three, there exist many more consensus algorithms.

In contrast to traditional centralized databases, a blockchain is **distributed** and retrievable from every participant of the network. Therefore, all transactions on the blockchain are transparent and visible to all participants in real-time. A blockchain removes the need of a Trusted Third Party (TTP) to secure and validate transactions within such systems and relies on a cryptographic proof instead. Thus, there is absolutely **no trust** in any entity of the decentralized system.

The blockchain technology can be seen — together with other emerging technologies such as Artificial Intelligence (AI), fog computing, and autonomous vehicles — as part of the current fourth industrial revolution. The blockchain technology, in particular, has the potential to radically change entire business sectors, governance, regulatory services, and society as a whole [15].

2.4 Related Work

The management of networks and services demands a multitude of methods, activities, procedures, and tools, with the final goal to ensure the proper functioning of the observed system. Such tools enable a network administrator to retrieve management information from the corresponding devices, analyze the obtained data and take decisions to optimize or repair a service. Within this workflow, information visualization can provide — if done right — a way to represent a large amount of data in a way perceivable much faster by the human user than the raw and often abstract data. As outlined by Colin Ware [17], good information visualization allows to perform cognitive work more efficiently and hence in less time. While scientific visualization addresses scientific data, information visualization addresses abstract data such as texts (e.g. configuration settings or log messages) and graphs (e.g. the VNFFGD described before or logical connections between different Internet Protocol (IP) addresses). Network services mostly provide this kind of abstract data.

Taking a look at related work in the field of information visualization applied to NFV environments, Guimarães *et al.* [5] discussed in their survey that Information Visualization should be explored on hot topics such as virtual networks as there was no publication addressing it up until the year 2016. Soles and Snider [18] examined specifically the process of virtualized network configuration, making it clear that there were no tools to assist the configuration, deployment, and testing of virtualized networks by 2016 either. Specifically, they found no single graphical tool for the creation of a network map directly from a configuration as given by the various descriptors in an SFC configuration. In other work, Franco *et al.* [6] presented *VISION*, a tool which provides interactive and selective visualizations for the assistance during NFV management. While the visualizations can help network operators to identify and alleviate problems in the context of VNFs, the *VISION* tool also provides a complete forwarding graph visualization. Although the tool can provide useful information about incorrect VNF placements or performance issues in the visualization, it is only focusing on already deployed services and previously configured monitoring systems and doesn't address the creation of new network services through the help of visualization tools.

In the context of SFC visualization, Sanz *et al.* [19] introduced SFCPerf, an automatic performance evaluation tool for SFCs. The tool ensures the repeatability of the performance measurements through the definition of a testing workflow, thus allowing the performance comparison among different SFC configurations based on the same test. The visualization module in their framework provides a user-readable interface to visualize throughput, round-trip time, and request rate of the given SFC. Based on their scenarios, they discovered that the main impact factors on the overall performance of an SFC were (i) the number of physical link hops between different nodes, and (ii) the competition for resources on shared physical nodes. These visualizations can be useful especially during the construction phase of an SFC while considering different topologies and NFV platforms, ensuring that the performance meets the desired requirements. In another work by Eichelberger *et al.* [20], *SFC Path Tracer* is presented, a troubleshooting tool for SFC environments, enabling the visualization of the trace of network packets in SFC domains. This trace generation is accomplished by mirroring probe packets as they traverse through the chain. Hence, SFC Path Tracer can be useful for the identification of problems within an SFC configuration, as it pinpoints the origin of a possible problem by providing packet trace information. The authors also argue that the tool could be expanded in the near future to a more general measurement tool (e.g. to support performance measurements as done by Sanz *et al.*).

Chapter 3

GENEVIZ

This chapter aims to outline the design and implementation of GENEVIZ (**Gene**ration, **V**alidation, and Visual**iz**ation of SFC Packages) — our proposed visualization tool. We will first cover the use case scenarios and conceptual architecture, followed by a brief discussion about the implementation of the GENEVIZ Prototype.

Although information visualization was already applied by a few works in the context of NFV and SFC environments, none of them addresses the process of SFC creation. We want to go a step further with this thesis, stating that a visualization tool could significantly simplify the construction of new SFCs from scratch by the composition of multiple VNFs, as well as the creation of new SFCs by the modification of existing ones. In general, an SFC can be visualized by nodes, with each node representing a single VNF, and edges, with each edges representing the traffic flow between two VNFs. Those elements build up a graph together, referenced to as *Forwarding Graph*. In the context of NFV, such a forwarding graph is called VNF Forwarding Graph, or VNFFG, and its respective descriptor is the VNFFGD. For each service described by an SFC, one needs in any case (i) zero to multiple VNFFGDs, (ii) one NSD, and (iii) one to multiple VNFDs, depending on the number of VNFs involved. Overall, there is a tremendous amount of data distributed over many files in order to construct and manage the properties of an SFC. GENEVIZ should be capable to handle this data through an interactive graphical user interface. In fact, a significant part of past work using information visualization — applied on network management — lacks in interactive features. Although the analyzed tools update information automatically, they are often focusing on the management itself and do not address the challenges concerning the creation and configuration of services [5]. Further, a visualization tool addressing SFC construction shall not be limited to the creation of completely new services from scratch, but also be able to handle the import of previously created services in order to create new services based on already existing ones.

3.1 Use Cases

Based on the lack of research addressing SFC construction with the help of a visualization tool, the use cases listed in Figure 3.1 were defined. We differentiated between User No.

1 and User No. 2 as the validation of an SFC Package is usually done by a second user, being a different actor than the creator of the SFC Package. Further, we tried to keep the use cases as abstract as possible for this subchapter and they can be seen independently from the architecture of GENEVIZ. The requirements defined here are directly used as a reference for the definition of the GENEVIZ architecture.

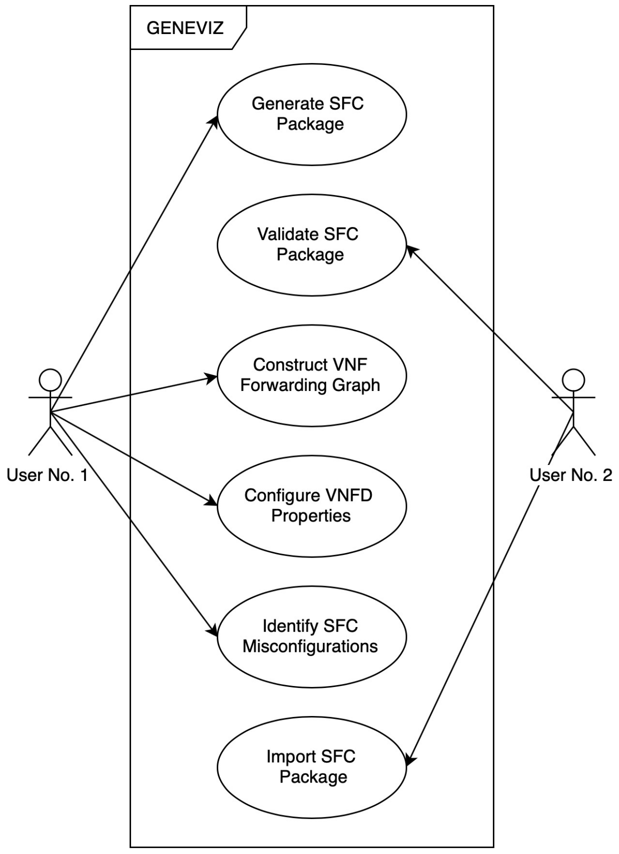

GENEVIZ
Generate SFC Package
Validate SFC Package
Construct VNF Forwarding Graph
Configure VNFD Properties
Identify SFC Misconfigurations
Import SFC Package
User No. 1
User No. 2

Figure 3.1: Use Cases of GENEVIZ

- **Generate SFC Package**: The user should be able to generate an SFC Package based on the configurations he applied through the usage of GENEVIZ. As this step is usually done at the end of the creation process, it depends on a valid construction of the graph. The SFC Package could also be stored remotely or be uploaded directly to a marketplace service during this step and is not limited to a downloadable file format. We see an *SFC Package* as an archive containing all the information required for managing the lifecycle of an SFC in a similar way a *VNF Package* holds the necessary information for a VNF.

- **Validate SFC Package**: Through GENEVIZ, one should be able to validate an already existing SFC Package in order to determine if the content of the package was modified in any way. Ensuring trust on an SFC Package is an important part during the deployment phase of new network service. An invalid SFC Package could indicate possible security concerns on it.

- **Construct VNF Forwarding Graph**: During the construction of a new SFC Package, the user should be able to define the path in which the traffic is flowing. This path is — generally spoken — a Directed Acyclig Graph (DAG) with each node only having one outgoing edge except the last node, which has none. The User Interface (UI) for this should be as intuitive as possible, allowing easy and quick adjustments of the forwarding graph.
- **Configure VNFD Properties**: Since the properties inside the VNF Descriptor are quite nested and handling with the file hierarchy is — especially for large and complex SFCs — time consuming and unnecessary, GENEVIZ should provide some graphical user interface, allowing to modify certain VNFD properties much easier and directly during the construction of an SFC.
- **Identify SFC Misconfigurations**: During the construction of a new SFC, recommendations should be provided to create better SFCs in terms of chaining. This can be especially helpful for not so experienced users constructing new network services.
- **Import SFC Package**: The last use case considers the import of a previously created SFC Package into GENEVIZ. By importing an existing SFC Package into GENEVIZ, the user can create a new network service based on the old one.

3.2 Architecture

The conceptual architecture of GENEVIZ is illustrated in Figure 3.2 and will be discussed in the following subchapters component by component. White blocks with a solid border represent internal components, grey blocks with a dashed border represent external components. The GENEVIZ platform can be divided into three main layers: *User Layer*, *Data Layer*, and *Blockchain Layer* respectively. Although the *Blockchain Layer* is not part of the GENEVIZ tool itself, it is an integral part of our proposed solution since it is needed for the validation of SFCs. GENEVIZ sets out mainly two goals: it aims (i) to simplify the process of creating new network services (i.e. SFCs) in general, and (ii) to ensure data integrity of previously created services by validating them.

While defining the architecture, we also considered the case of creating large and complex SFCs where the same VNF is used multiple times but with different configurations. Hence, in GENEVIZ, we differentiate between a *VNF Template*, being the original and "generic" VNF Package which was imported through the *User Layer*, and a *VNF Package*, which is the VNF being part of the composed SFC and thus an instance of the template. A *VNF Package* might have customized VNFD properties which are different from the *VNF Template*. In the same meaning, we can speak of an *SFC Template* on the *User Layer* and of an *SFC Package* on the *Data Layer* when talking about SFCs in the subsequent chapters.

The Data Flow of GENEVIZ can be described as follows: The end user accesses the platform through the *User Interface* of the web application. The *User Interface* provides several interactive visualizations, which depend on data provided by the *Visualization Manager*. An integral part of this data is given by the *Template Catalog*, which retrieves

the templates from the *Templates Collector*. The Collector on the other hands gets its data from multiple different sources (e.g. marketplaces, independent catalogs, or a simple manual upload from the local machine) and can be extended if more sources seem to fit into GENEVIZ.
While creating an SFC through the *Service Constructor* visualization, the VNF Templates from the *Template Catalog* are transmitted through the *Visualization Manager* to the *Management API* and then stored through the *Package Handler* on the *Packages Database*. If an SFC Package is generated, the *SFC Package Generator* creates an SFC Package based on the information provided from the *Visualization Manager* (e.g. the list of included VNFs, the forwarding graph, and some properties for the NSD) and transmits the package to the *SFC Package Manager*. The latter sends it to the *Package Handler*, where it is sent further through the *Management API* and the *Visualization Manager* back to the *User Interface*, where it is downloaded through the web application.
Both the *Placement Recommender* visualization and the *VNFD Editor* visualization need the current VNFD file from the VNF Package being part of the SFC in order to do their job. However, as the packages usually are uploaded as ZIP files or another compressed file format, the VNFD of the VNF Package currently being added to the SFC needs to be retrieved after the upload from the *Packages Database* through the *Package Handler* and the *Package Parser*, before sending the VNFD back through the *Management API* to the *Visualization Manager*.
In the case that the properties from the VNFD are edited through the *VNFD Editor* visualization, the updated VNFD file is transmitted through the *Visualization Manager* to the *Management API*. The corresponding VNF Package is then retrieved from the *Packages Database* by the *Package Handler* and then transmitted to the *Package Parser*, which extracts the ZIP file in order to change the properties in the VNFD file. The updated VNF Package is then stored back into the Packages Database.
For the validation or import of an already existing SFC Package, the package is uploaded through the *User Interface* and then transmitted through the *Visualization Manager* to the *Management API*, where it is sent further through the *Package Handler* to the *Packages Database*, to be finally stored there. For the validation, the *SFC Package Validator* communicates over the *SFC Package Manager* with the *Blockchain API* from the *Blockchain Layer*. The API then tries to retrieve the corresponding hash from the data property for the given transaction ID. For the import of an SFC into the GENEVIZ tool, the *Package Handler* gets the SFC Package from the *Packages Database* and extracts the content with the help of the *Package Parser* in order to prepare and send the necessary information for the visualizations back to the *Visualization Manager* through the *Management API*.

The general architecture of GENEVIZ is composed of completely separate, but interconnected components. This modularization mechanism allows to replace existing modules or add new modules without affecting the remaining components of the GENEVIZ platform. We imagine GENEVIZ as a completely local application with both *User Layer* and *Data Layer* running on the local machine. Hence, no trust is necessary in the provider of a component of GENEVIZ from the end user perspective. Nevertheless, through the modularization, specific components could also be located remotely. In the following subchapters, the components from each layer of GENEVIZ are discussed in greater detail.

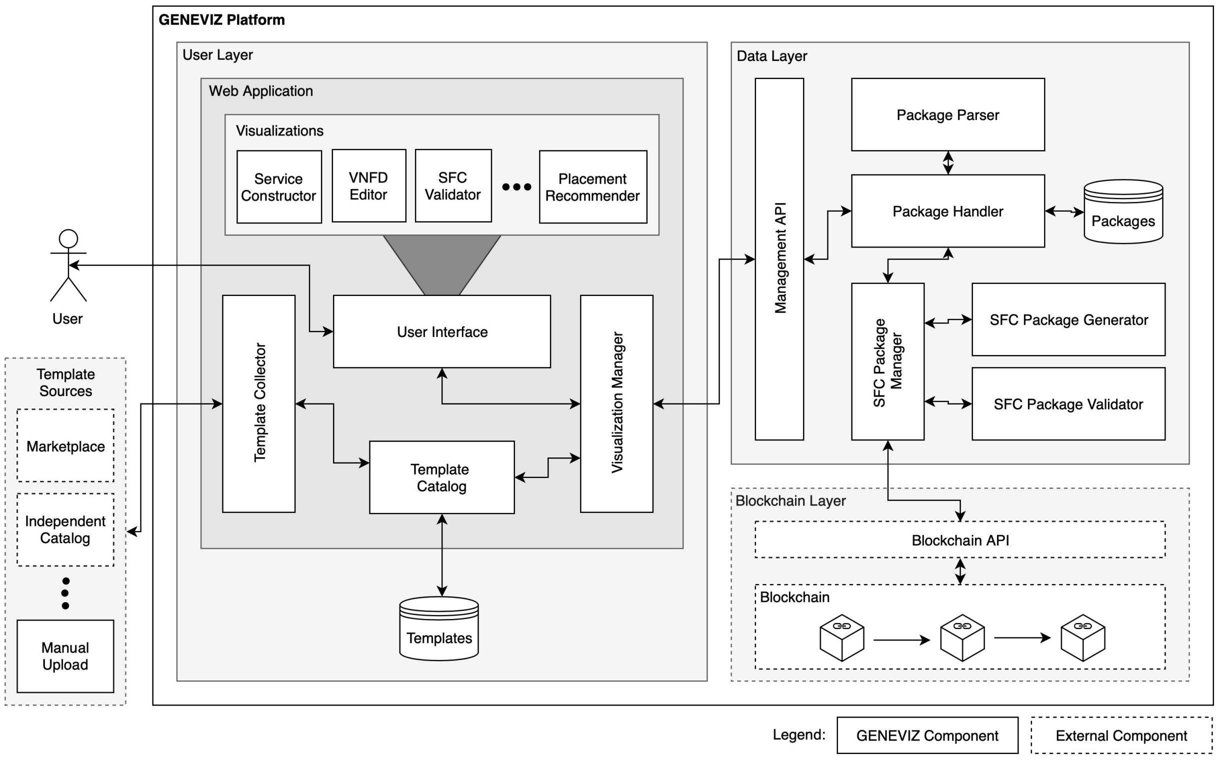


Figure 3.2: Conceptual architecture of GENEVIZ

3.2.1 User Layer

The User Layer contains the elements responsible for the interaction between the user and the GENEVIZ platform. Except for the *VNF Templates Database*, all other components on this layer build up together the *Web Application* of GENEVIZ. We decided to propose a web application on this layer as there is no need to download and install specific software for the end user, as long as the assumption, that the currently installed browser is supporting the GENEVIZ web application, holds.

Template Collector

As VNF and SFC Templates can come from several sources, this component has the job to communicate with other platforms in order to gather the set of templates available for the SFC construction. These platforms can include several marketplaces, independent catalogs or the simple manual upload of a ZIP file. VNFs and SFCs purchased over a marketplace through a specific user account could be retrieved during the initialization of the platform, if authorization is provided, and could hence directly be integrated within GENEVIZ. VNF Templates are needed in order to construct new network services from scratch. SFC Templates can be validated and imported in order to create new SFCs based on previously created SFCs.

Template Catalog and Database

The *Template Catalog* handles first the communication between the *Template Collector* and the *Templates Database*, with the latter holding all templates collected during the launch (or later in the lifecycle) of the application. As the data stored in this database is not needed for the SFC Package construction, we clearly separate it from the *Packages Database* located on the *Data Layer*. The collector transmits the templates to the catalog, which are subsequently stored on the database. Further, the catalog retrieves all stored templates in order to provide them to the *Visualization Manager* for its usage. In fact, the storage of the templates in GENEVIZ is only mandatory for the manual upload. VNFs collected through the integration of a marketplace or independent catalog don't necessarily need to be stored on a database inside GENEVIZ, and could instead be directly transmitted through the *Visualization Manager* to the *Data Layer*, where they will be stored as *VNF Package* respectively if they are added to the SFC currently in draft. The consideration of both VNFs and SFCs in the catalog can be especially interesting if we consider the case that a previously created SFC from a marketplace could be imported into GENEVIZ, which would afterward be extended in a next step by adding additional VNFs from a catalog.

Visualization Manager

This component is the key bridge between the *User Layer* and the *Data Layer* as it handles the communication between the mentioned two on the *User Layer*. The *Visualization Manager* is also in charge of handling the interaction between the user and the visualizations, which also includes the provisioning of necessary data needed for the visualizations, such as the catalog from the *Template Catalog* component. Hence, this component can also be seen as the ”store” (i.e. current state) of the web application. From a Model-View-Controller (MVC) perspective, the *Visualization Manager* holds both the Model and the Controller, leaving the *User Interface* as a dumb *View* component.

User Interface

The web-based *User Interface* provides several visualizations to the end user but is not limited to them exclusively. This component is the only way where an interaction between the user and the GENEVIZ platform can occur. It should allow an intuitive and easy way for the end-user to interact with the visualizations and the rest of the user interface. It's important to note here that in our context, the term *Visualization* is used in a much broader sense, not only referring to the visual representation of abstract data but also being an interactive interface providing forms to modify the data. As the visualizations are modular as well, they could be adapted to the network operator's needs. We currently propose four visualizations for our solution, although a lot more could be integrated into GENEVIZ:

1. **Service Constructor** — By chaining two or more VNFs, a forwarding graph is created, defining the flow in which the traffic is traversing through the VNFs. The

graph should be constructed with the help of this visualization and is based on the end user requirements for the new network service. Hence, the *Service Constructor* should visually represent the whole SFC.

2. **VNFD Editor** — The properties of the VNFD should be configurable in an easier way than navigating through the file system in order to find the respective VNFD, open it in a separate Editor, and adjust the properties. For GENEVIZ, we decided that the properties in terms of hardware allocation could be some of the most important ones during the creation of a new network service. Hence, (i) Number of CPUs, (ii) Memory Size, and (iii) Disk Size should be adjustable in the VNFD Editor, which can be opened for every VNF Package added to the SFC currently in draft. This can be especially interesting for large SFCs, where multiple VNF Packages from the same VNF Template, but with different hardware properties, are defined. It's important to denote here that the VNFD Editor is not limited to the three properties addressing hardware allocations, but could involve any properties from the VNFD.

3. **SFC Validator** — To check the integrity of an already created SFC, this visualization should visually represent the state of the validation of the SFC Template after it has occurred.

4. **Placement Recommender** — For GENEVIZ, we propose that this visualization should go hand in hand with the *Service Constructor* visualization, but it is not limited to it and could be completely independent for other SFC visualization tools. Further, the *Placement Recommender* could combine several different approaches together under one single visualizations. To make a universal statement in terms of general performance for a given connection, there doesn't seem to exist enough data from current NFV environments. But as the placement and order of VNFs can have a direct impact on the whole performance of an SFC, we see the *Placement Recommender* as an advisor in terms of *Chaining of VNFs*. Hence, this visualization should either recommend, not recommend, or stay neutral on a currently created connection between two different VNFs. For this, we propose the extension of the current ETSI specification of the VNF Descriptor by adding two new properties, namely `target_recommendation` and `target_caution`. The `target_recommendation` should list services (i.e. VNFs) which are *recommended* as a target. Hence, an edge from the current VNF to the one mentioned in the recommendation list is advisable. The `target_caution` should list services (i.e. VNFs) which are *not recommended* as a target. The service names in both of the new properties should refer to the `service_types` property of the target VNF. During the construction of the forwarding graph, the *Placement Recommender* should visualize these placement suggestions in order to guarantee *the best chaining* for the SFC.

3.2.2 Data Layer

In general, the *Data Layer* can be seen as the backend of the GENEVIZ platform, as it is mostly responsible for data storage and no interaction with the user finds place here. It is also responsible for the communication with the *Blockchain Layer* in order to handle the validation of already existing SFC Packages.

Management API

This component handles the communication between the *User Layer* and the *Data Layer* on the latter side. Specifically, it handles requests between the *Visualization Manager* and the *Package Handler* such as the transmission of VNF and SFC Packages as well as the VNFD file for example.

Package Parser

Since VNF and SFC Packages consist of multiple files and folders with a quite nested hierarchy and are usually compressed as a ZIP file or another compressed file format, the *Package Parser* has the job to extract those files in order to make it accessible to the *Package Handler* component. It parses the VNFD file from a VNF Package, which is needed for several visualizations on the *User Layer*, as well as the NSD file, which is needed for the import of an existing SFC Package.

Package Handler

The *Package Handler* transmits either VNF or SFC Templates from the *Management API* to the *Packages Database* to be instantiated as a VNF or SFC Package, SFC Packages from the *SFC Package Manager* to the *Management API*, VNF or SFC Packages from the Database to the Parser, or Data from the Parser to the *Management API*. This component is also in charge of updating the VNFD properties as supported through the *VNFD Editor* visualization.

SFC Package Manager

Handles the communication between the *Data Layer* and the *Blockchain Layer* by sending a request to the *Blockchain API* to either store a hash or validate an SFC Package. Also, this component triggers the generation and validation of an SFC Package by communicating with the respective two components. The SFC Package generated by the *SFC Package Generator* is sent back further through the *Package Handler* to the *Management API*, to then sent to the *User Layer*. Further, the results from the validation are transmitted through the *Package Handler* to the *Management API*.

SFC Package Generator

As its name says, this component generates the SFC Package by aggregating the different VNF Packages being part of the SFC and creating the NSD by the given data provided by the *User Layer*. In general, the VNFFGD is also part of the SFC Package, but is currently not intended to be generated by GENEVIZ within this component, since the VNFFGD depends too heavily on the NFV environment. Open Baton and OpenStack, for example, have different ways to describe the VNFFGD. Optionally, the *SFC Package Generator* also requests the writing of the hash of the generated SFC Package on the blockchain and receives a transaction key after successful writing on the blockchain. This transaction key is stored together with the account address on the SFC Package and used for the later validation of this package.

SFC Package Validator

This component is responsible for the validation of already created SFCs. By comparing the hash of the SFC Package, together with a given transaction key, with the hash written on the blockchain for this transaction key, GENEVIZ can check if the package content matches the initial one from the creator of the SFC Package. For GENEVIZ, we differentiate between three different possible states, where we see an SFC Package after its validation:

(i) **Valid** — If the hash matches the one written on the blockchain for the given transaction key, data integrity of the package content could be established as the content was not modified in some way and is the original one from the moment of creation.

(ii) **Invalid** — As the SFC Package was somehow modified, the hash of the downloaded package doesn't match anymore with the one written on the blockchain for the given transaction key. Hence, the package becomes invalid and data integrity could not be established.

(iii) **Unknown** — During the creation of a new SFC Package, the user can decide to not store the hash of the package on the blockchain due to some reasons. Thus, there is no transaction key or address given for the package, which can be used for verification. In this case, GENEVIZ doesn't want to make a statement about the integrity of the content. It can either be that the content was modified or stayed the same, so GENEVIZ marks the package as *Unknown*.

3.2.3 Blockchain Layer

Although this layer is not part of GENEVIZ itself, it can be seen as a part of the GENEVIZ platform as a whole. We could imagine any kind of blockchain for this layer, if it supports some data property inside a single transaction where we can store the hash value of our created SFC Packages, such as Ethereum, Bitcoin or IOTA can do it. As GENEVIZ aims to be used locally with no centralized server for the uploaded VNF Packages or generated

SFC Packages, we think a blockchain would fit best for this kind of data storage. A centralized database needs to be maintained somewhere — ensuring trust on it from the perspective of a network operator seems to come with a certain risk. Thus, we imagine a decentralized blockchain for the storage of the data hashes of the SFC Packages, accessible from any local machine as long as a connection can be established between the computer and the blockchain. Further, in the work of [15], the usage of a blockchain for a digital asset's proof-of-existence was discussed. We would like to build upon that, using the hash of an SFC Package as an immutable time-stamp to certify the authenticity of a created package. By storing the fingerprint of the package — during its generation — on a publicly accessible blockchain, a proof-of-existence can be made at a later point in time on any machine running GENEVIZ.

Blockchain API

This component handles the communication between the *Data Layer* and the *Blockchain Layer* from the side of the latter one. As we imagine any kind of blockchain supporting some data property on the transaction itself, this API will be different for a different blockchain. For GENEVIZ, we could also imagine an API providing interoperability on different blockchains at the same time through one single interface. Hence, the user could choose on the *User Layer*, on which blockchain he wants to store the hash. Such an *Interoperability API* can be seen in the work of [21], where a solution for storing and retrieving data on seven different blockchains is provided.

3.3 Prototype and Implementation

We implemented the GENEVIZ Prototype using HTML, CSS, and JavaScript on the *User Layer* and Python on the *Data Layer*. The prototype was implemented considering the requirements defined in Chapter 3, serving as a Proof of Concept (PoC) for the conceptual architecture of GENEVIZ. As we tried to hold the application as light as possible, no database — as it was proposed in the general architecture — was integrated. Instead, the *Web Application* on the *User Layer* holds its data through a single JavaScript object, while on the *Data Layer* all files are stored in a temporary folder on the local machine, also eliminating the need for a database integration.

3.3.1 User Layer

For the *Web Application* on the *User Layer*, we decided to go with React 16.8.3[22], a JavaScript library for building User Interfaces which gained wide popularity also for large applications in recent years (e.g. Facebook and Instagram are using React for their web applications) [23]. As React itself is not a framework, the state management of the application is handled with Redux 4.0.1 [24], a JavaScript library for handling global state management. Redux doesn't depend on React and can be used together with any UI library. The source code of the web application is mainly written in TypeScript 3.3.3333 [25], a strict syntactical superset of JavaScript, adding static typing to the language and using a bunch of JavaScript libraries provided through the Node Package Manager (NPM) [26]. The TypeScript source code hence needs to be compiled to plain JavaScript code, which is supported by the majority of today's web browsers. As this type of compilation from TypeScript to JavaScript is converting the source code on a similar level of abstraction, one often refers to this as *transpilation*. For this, Babel 6.26.3 [27] in combination with Webpack 3.8.1 [28] is used, with the latter aggregating the different files into one single and compressed JavaScript file.

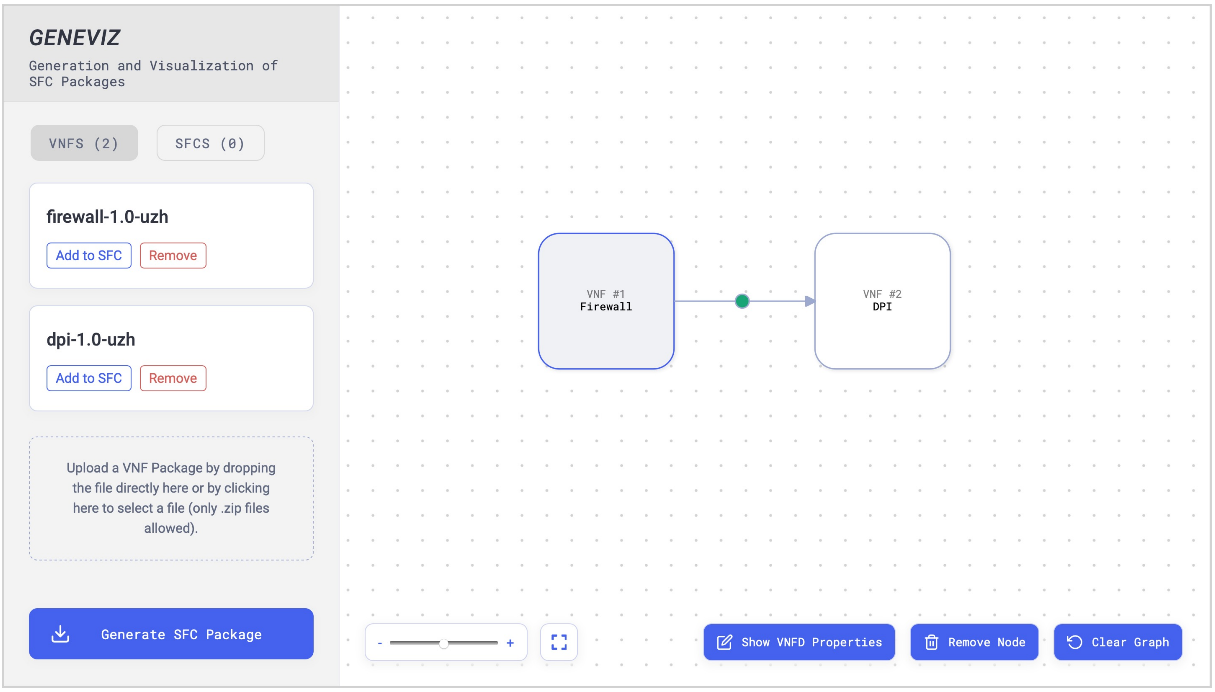


Figure 3.3: Screenshot of the GENEVIZ Prototype

A screenshot from the web application of the GENEVIZ Prototype can be seen in Figure 3.3. On the left side, there is a menu, allowing the user to manually upload zipped VNF and SFC Packages by using the dropzone with the dashed border. By clicking on the respective buttons for "VNFS" and "SFCS", the user can switch between the two catalogs. For the GENEVIZ Prototype, only ZIP files are allowed to be uploaded.

At the bottom of the menu, a blue button allows the generation of an SFC Package, based on the constructed SFC. This button only appears if the constructed SFC is valid. Hence, the button is not visible at the initial start of the application, as an empty SFC is seen as invalid. The requirements for an SFC to be valid are at least two VNFs connected through one edge.

The User Interface shows an alert on the top right corner if any kind of error message should be passed down to the user. This can especially be helpful during the graph construction or import of new packages, as there is some kind of feedback from the web application if some action fails or is not allowed. Examples for this could be a wrong format of the VNF Template (e.g. the *Package Parser* can't find the VNFD) or the attempt to create a loop during the forwarding graph construction.

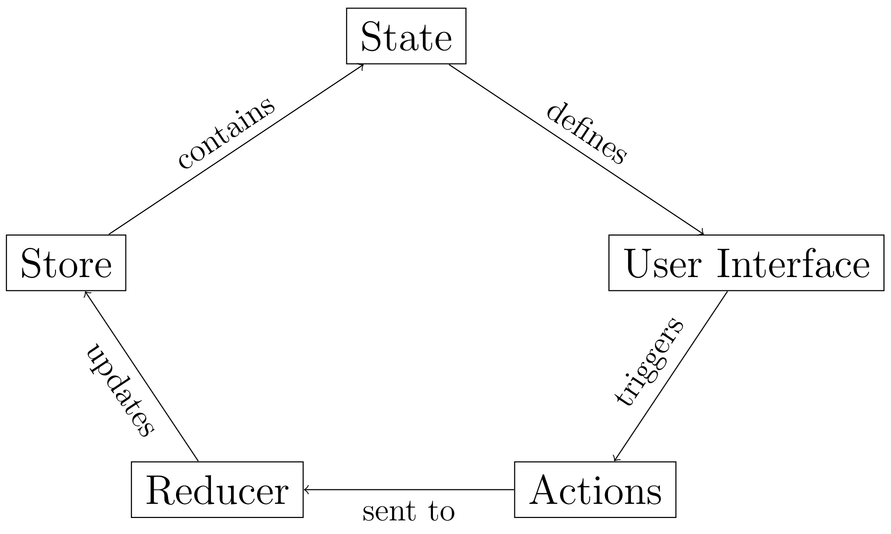


Figure 3.4: Software Architecture of GENEVIZ with the Redux pattern

For the GENEVIZ Prototype, we assume the VNFD to lie in the standardized path `/Descriptors/vnfd.json` inside the VNF Package. The VNFD is retrieved after a VNF is added to the SFC and not during the manual upload of a package, hence an error message for the wrong path is not returned before the addition of the VNF. Any other location for the VNFD is not supported for the prototype. This location could be changed in the source code or even dynamically be set in order to handle other paths. We don't actually care for the rest of the content of the VNF Package as the only necessary file for our visualizations is the VNFD.

As already mentioned in the previous subchapter, the VNF Forwarding Graph Descriptor cannot be generated through the GENEVIZ Prototype as it depends too heavily on the NFV environment. For the forwarding graph of the VNFs, under the assumption that there is only one single directed path going through each VNF once, a simple list is enough to represent the forwarding graph of the network traffic. For the GENEVIZ Prototype, we decided to define this list through the VNFD reference inside the `vnfd` property of the NSD (e.g. the first VNFD path refers to the first VNFD of the SFC, the second VNFD to the second VNF, and so on).

The name of a template is received through its file name. Thus, the file `firewall-1.0-uzh.zip`, for example, is named as "firewall-1.0-uzh" in the templates list in the menu on the left side of the GENEVIZ application. For the title of the nodes, we extract the `name` property from the VNFD file and set the nodes according to the name specified in the VNFD, which we don't have at the moment of the ZIP file upload in the templates list and is the reason we rely on the file name for the templates.

As the Frontend is implemented with React and Redux, we briefly want to discuss the Redux pattern as seen in Figure 3.4, since it sets the main software architecture of the web application. The whole state of the GENEVIZ web application is stored in an object tree inside a single *Store* object and the only way to change this state tree is to trigger an *Action*, an object which describes what happened and usually also contains some data necessary for the state transition. To define how the different actions alter the state tree, a

Reducer is needed, eventually transforming the previous state to the next one. This state change triggers the re-rendering of the *User Interface*, which is written by using React. A Reducer can be seen as a pure function that takes the previous state and an action and returns the next state. Hence, the Reducer is similar to a higher-order function, a functional programming concept.

Visualizations

The visualizations of the GENEVIZ Prototype refer back to both the conceptual architecture of GENEVIZ as seen in Figure 3.2, as well as the use cases defined in Chapter 3.1 and also shown in Figure 3.1.

Service Constructor In general, the *Service Constructor* can be referenced to as the right side of the GENEVIZ Prototype, consuming about 2/3 of the application window, also seen in Figure 3.3. It depends on the addition of VNF Templates from the left menu, consuming about 1/3 of the application window. After successfully uploaded a VNF Package through the Dropzone in the left menu, the VNF Template can be added to the SFC as a VNF Package by clicking on the blue bordered "Add to SFC" Button and removed from the templates list by clicking on the red bordered "Remove" button. After clicking the blue "Add to SFC" button, a node appears in the *Service Constructor* visualization. If a node is selected by clicking on it, three blue buttons appear at the bottom right corner of the application window:

- **Show VNFD Properties** — Opens the VNFD Editor
- **Remove Node** — Removes the selected VNF Package from the SFC and hence also from the graph. Any other connection with other nodes is detached by removing the corresponding edges as well.
- **Clear Graph** — Removes all nodes and edges from the graph and resets the entire SFC to an empty state.

If an edge is selected by clicking on it, the first button ("Show VNFD Properties") is not shown, as no VNFD Properties can be edited on an edge. The graph construction tries to be as strict as possible, not allowing any loops or multiple outgoing connections from the nodes.

Nodes can be relocated by selecting them first and then moving them around with the mouse cursor. With this, the nodes can be rearranged in order to construct a clearly ordered SFC, which can be helpful especially for large and complex SFCs.

A new edge can be created by first pressing and holding the *Shift* key on the keyboard, then selecting the first node with the mouse cursor by clicking on it, moving the mouse cursor to a second node which should be connected, and then releasing both the mouse button as well as the *Shift* key. While doing all this, an arrow should appear on the graph plane, making the connection visible as a preview before the actual edge is created. With this, the different VNFs can be connected to each other in order to define the VNF Forwarding Graph. The type of edge depends on the *Placement Recommender* visualization.

VNFD Editor To open the *VNFD Editor*, a node must be selected from the *Service Constructor* visualization. When a node has been selected, the blue ”Show VNFD Properties” button appears at the bottom right corner of the GENEVIZ application window beside two other blue buttons. Clicking on the respective button opens up a popup window, which represents the *VNFD Editor* visualization. For the GENEVIZ Prototype, we expected each VNF has *at least* the same three hardware properties in their descriptor. The hardware properties are assumed to be located in the VNFD as follows:

```
    "vnfd": {
        "attributes": {
            "vnfd": {
                "topology_template": {
                    "node_templates": {
                         "VDU1": {
                            "capabilities": {
                                "nfv_compute": {
                                  "properties": {
                                    "num_cpus": 1,
                                    "mem_size": "4 GB",
                                    "disk_size": "5 GB"
                                  }
                                }
                         }
                    }
                }
            }
        }
    }
```

Thus, the three properties `num_cpus`, `mem_size`, and `disk_size` can be edited through the *VNFD Editor* and saved on the VNFD by clicking on the blue-bordered ”Apply Changes” button. When clicking on the ”Cancel” button, the current changes are not saved on the VNFD.

Placement Recommender Before a new edge is being placed between two nodes on the graph plane, the edge type is chosen based on the two properties `target_recommendation` and `target_caution` inside the VNFD of the source node (i.e. the *Source VNF*). Based on this information, the edge holds a green dot for a *recommended* target VNF, a red dot for *not recommended* target VNF, and a blue/white for neutral targets. The latter is especially important, as a VNF does not hold the entire list of all possible VNFs as targets in its properties, leaving the statement about the placement recommendation neutral.

SFC Generation Although the generation of an SFC Package itself doesn't have or need a visualization, there is certain information necessary before the generation of a new SFC Package can happen. We decided to provide the setting of three properties from the NSD — namely `name`, `vendor`, and `version` — within this step. By clicking the "Generate SFC" button, a popup window is opened where the properties can be set. Further, by clicking the checkbox, two additional properties appear for the writing of the hash of the SFC Package on the blockchain, namely `address` and `privateKey`, which are both needed to store the hash on a given address and sign it with the private key.

SFC Validator The validation of an uploaded SFC is triggered directly during the upload and doesn't need to be done manually. In the case, the validation succeeds, a green "Valid" label appears below the SFC Template on the left menu. If the validation fails, a red "Invalid" label appears. A grey "Unknown" label appears for the unknown integrity of an uploaded SFC Package.

3.3.2 Data Layer

For the *Data Layer*, we chose Python 3.7.0 [29] together with Flask 1.0.2 [30]. Flask is a micro web framework as it does not require particular libraries or tools itself. We used Flask's `route()` decorator to bind a function to a URL and hence implementing the *Management API* with the required functionality. The *User Layer* communicates with the *Data Layer* by using the Fetch API, a Web API for fetching resources, through JavaScript. The Fetch API allows making a network request similar to an XMLHttpRequest (XHR), with the important difference that the Fetch API is using *Promises*, an object representing the eventual completion or failure of an asynchronous request and its resulting value. Appendix A provides the complete list of the HTTP Endpoints from the *Management API* with the corresponding parameters needed for the requests and the different responses.

All the necessary components from the *Data Layer* of GENEVIZ are integrated into the *Management API* source code, which hence handles ZIP file storage, extraction, and parsing, as well as the SFC Package generation and validation.

Figure 3.5 shows the file structure of an SFC Package generated by the GENEVIZ Prototype. For the prototype, we defined an SFC Package to be a ZIP file, named as `sfc-package.zip`. This ZIP file acts as a "wrapper" ZIP file for the actual `sfc.zip` file. The `sfc.zip` file contains several VNF Packages, with each VNF having its own separate folder for its respective content. If an SFC Package has multiple VNFs from the same *VNF Template*, there is still just one folder, as the content, except for the VNFD, stays the same. Hence, for an SFC Package with multiple VNFs from the same template, we have multiple VNFDs lying in the same `Descriptors` folder, but with different UUIDs. This `sfc.zip` file has — besides the different VNF Packages — another file named `nsd.json`, which represents the Network Service Descriptor of the SFC. At the same hierarchy level as the `sfc.zip` file, there is another file named `geneviz.json`, containing two properties, namely `txHash` and `address`. The transaction hash property `txHash` is retrieved from the blockchain itself after the hash is written on the blockchain, and is known as the transaction ID of the transaction for the Ethereum Blockchain. The `address` is given by the user itself when the generation of the package is requested. The `geneviz.json` is therefore needed for the validation of an SFC Package as it contains the transaction key necessary for the lookup of the data properties for this transaction key. If the data property retrieved from the transaction matches with the computed hash from the content of the `sfc.zip` file, the package is seen as *Valid*. Does the hash found on the blockchain for the given `txHash` not match with the computed hash from the content of the `sfc.zip` file, the package is seen as *Invalid*. If both the `txHash` and the `address` properties are empty strings, GENEVIZ has not stored any hash on the blockchain for this SFC Package and thus the data integrity of the SFC Package is seen as *Unknown*.

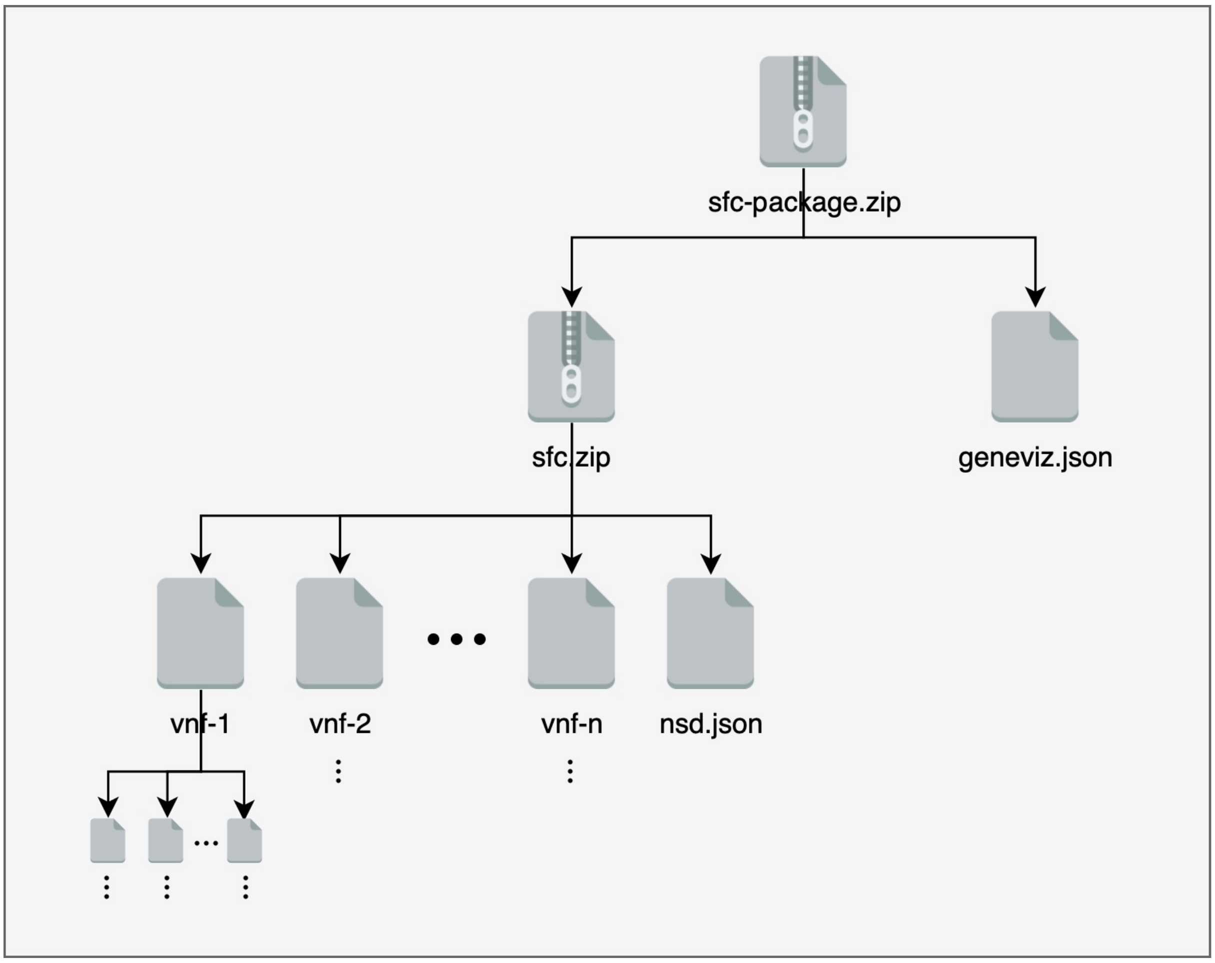
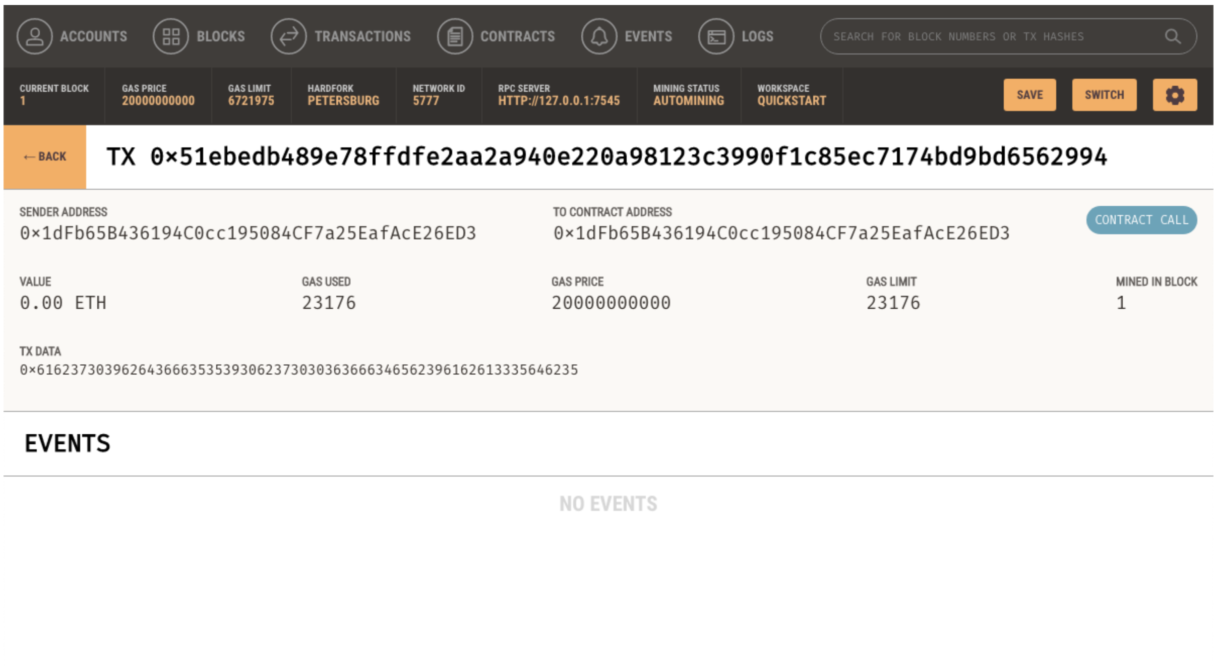


Figure 3.5: File Structure of SFC Package for the GENEVIZ Prototype

3.3.3 Blockchain Layer

The *Blockchain Layer* of the GENEVIZ architecture proposes the usage of a *Blockchain API* in order to communicate with any underlying blockchain supported by the API. We decided to take a more simplified version of this API and are using parts of the source code provided by [21] to communicate with the *Ethereum Blockchain* [31]. Hence, we extracted the majority of the source code from the Ethereum Adapter and adopted it for our use case. For easier testing purposes and in order to run GENEVIZ fully locally, we used Ganache [32], a personal blockchain for Ethereum development, which can run on the local machine and connected through the Endpoint URL provided by Ganache with GENEVIZ. The *Ethereum API* of the GENEVIZ Prototype takes usage of web3 [33], a python implementation of web3.js. Both web3 and web3.js are Ethereum APIs connecting to the Generic JSON RPC of Ethereum. For our prototype, we directly integrated the *Blockchain API* into the *Management API* by importing the former into the latter one. Hence, communication between the two through an API endpoint becomes obsolete.

Figure 3.6 shows a screenshot of a successful transaction on the Ethereum Blockchain by using Ganache. The property "TX Data" in the beige box contains the necessary data, which is then translated by the *Blockchain API* to the hash of the `sfc.zip` file, to then be further used for the validation of the package.



CURRENT BLOCK 1 | GAS PRICE 20000000000 | GAS LIMIT 6721975 | HARDFORK PETERSBURG | NETWORK ID 5777 | RPC SERVER HTTP://127.0.0.1:7545 | MINING STATUS AUTOMINING | WORKSPACE QUICKSTART

TX 0×51ebedb489e78ffdfe2aa2a940e220a98123c3990f1c85ec7174bd9bd6562994

SENDER ADDRESS
0×1dFb65B436194C0cc195084CF7a25EafAcE26ED3

TO CONTRACT ADDRESS
0×1dFb65B436194C0cc195084CF7a25EafAcE26ED3

VALUE	GAS USED	GAS PRICE	GAS LIMIT	MINED IN BLOCK
0.00 ETH	23176	20000000000	23176	1

TX DATA
0×6162373039626436663535393062373030363666346562396162613335646235

EVENTS

NO EVENTS

Figure 3.6: Screenshot of Ganache

Chapter 4

Evaluation

In order to validate the usability and technical feasibility of GENEVIZ as we proposed, the following sections describe three case studies, focusing on three different use case scenarios. The case studies aim to cover the different visualizations provided by GENEVIZ, showing their helpfulness in the context of SFC construction and validation. In the first case study, we discuss both the construction of a new network service (i.e. an SFC) by using the *Service Constructor* visualization, as well as the usefulness of the *Placement Recommender* visualization, with the latter helping us to better optimize our newly constructed network service in terms of chaining. In the second case study, we generate an SFC Package by using the constructed SFC from Case Study No. 1 and also trigger the writing of the hash of the SFC Package on the Ethereum Blockchain. In the third case study, we take a second user in, perform a validation of the created SFC Package from Case Study No. 2., import it into GENEVIZ in order to adjust some properties of an associated VNF Package, and conclusively generate a new SFC Package based on the imported one.

4.0.1 Case Study No. 1 — Construction of an SFC

Addressing the most prominent visualization inside GENEVIZ, namely the *Service Constructor*, let's consider a user with the specific demand to create a new network service, which should finally be deployed in an NFV environment in order to fulfill its purpose. For this, the user bought — beforehand — three different VNF Packages from an external source (e.g. from a marketplace), which are needed in order to create the SFC. Concretely, the user bought the following three VNF Packages: a *Deep Packet Inspection* (DPI) VNF, a *Firewall* VNF, and a *Load Balancer* (LB) VNF.
In a first step, the three VNFs are imported via manual upload in the menu on the left side of the GENEVIZ web application. After uploading them, the *VNF Packages* appear in the left menu as *VNF Templates* since they could be added multiple times for the same SFC. By clicking the blue-bordered "Add to SFC" button for each VNF Template once, each template is added as a VNF Package to the SFC and appears as a squircled node on the graph plane on the right side of the web application.
Now, the user makes the first connection between two VNFs by holding the *Shift* key on the keyboard, clicking on the DPI node, holding the left mouse button, moving the cursor

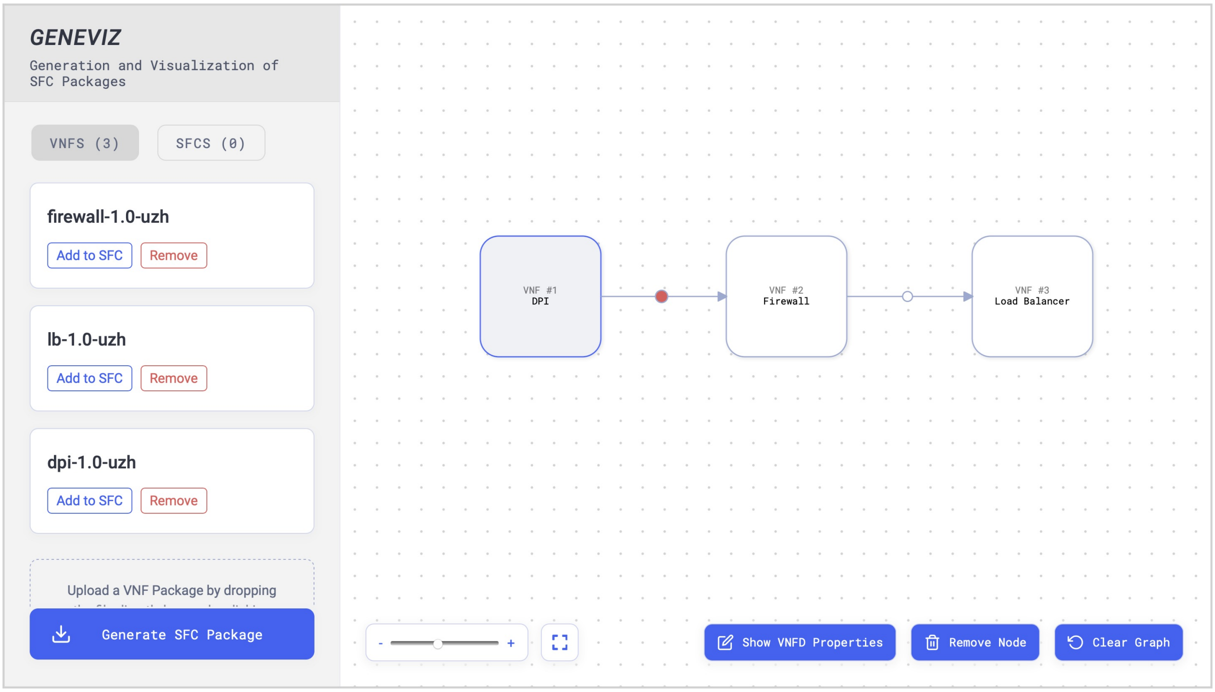


Figure 4.1: Screenshot 1 of GENEVIZ for Use Case No. 1

to the Firewall node, and releasing both the *Shift* key as well as the mouse button. This created an edge between the DPI and the Firewall node. By connecting the Firewall with the LB node with the same approach, the second edge is created. The current graph of this construction can be seen in Figure 4.1.
This first draft of the SFC can be seen as a misconfiguration, although it would not be wrong to create an SFC like this. The current construction also makes it clear that the user is not very experienced with the creation of SFCs since a red dot on the edge between the DPI and the Firewall node appears. This red dot is part of the *Placement Recommender* visualization of the GENEVIZ Prototype and indicates that the connection is not recommended for an SFC. Hence, the user overthinks the current construction and swaps the DPI and the Firewall node by deleting the two created edges, then swapping the position of the DPI and the Firewall node, and then connecting the three nodes again by creating two new edges. Now. the first edge even has a green dot, indicating a recommended connection. The graph of this enhanced construction is shown in Figure 4.2.

4.0.2 Case Study No. 2 — Generation of an SFC Package

This case study addresses the generation of the SFC Package and assumes a correct construction of the forwarding graph in the graph plane as given in Case Study No. 1. An SFC such as the one constructed in Figure 4.2 is valid, and hence a blue button with the label ”Generate SFC Package” appears at the bottom left corner. For this case study, we consider the exact same SFC as constructed in Case Study No.1 as a base for the subsequent steps.
By clicking on this blue button, a popup window appears, requesting the user to define the name, vendor, and version for the SFC Package. Our user sets these three properties

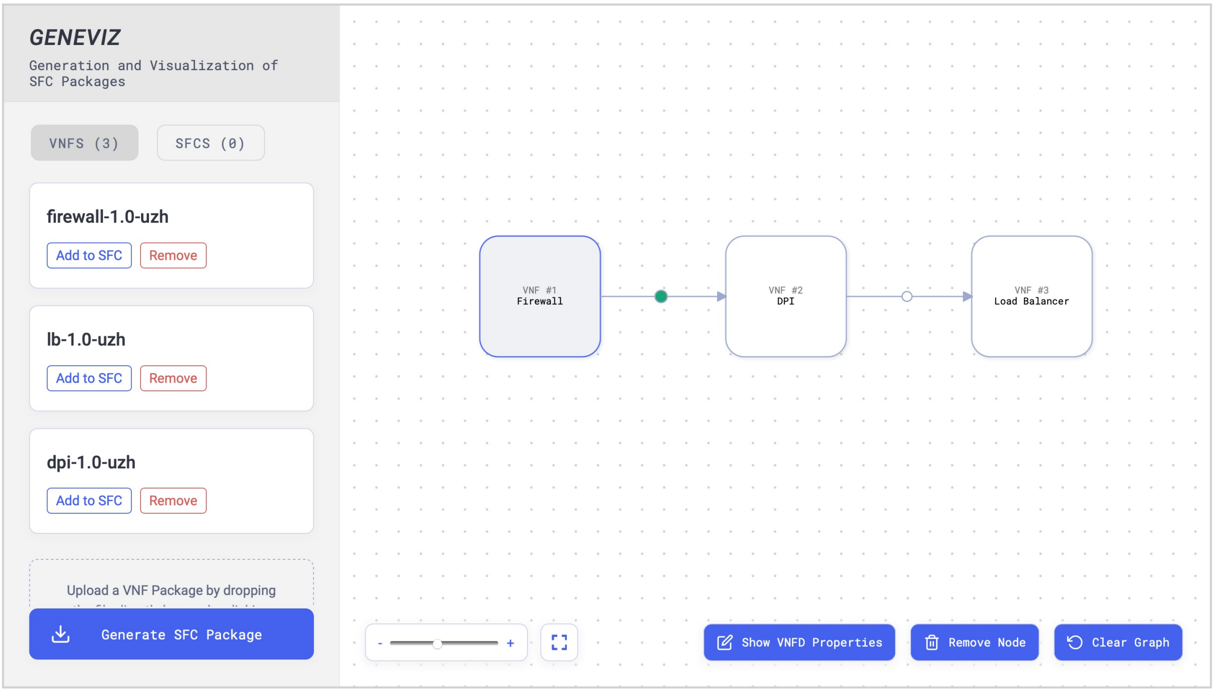
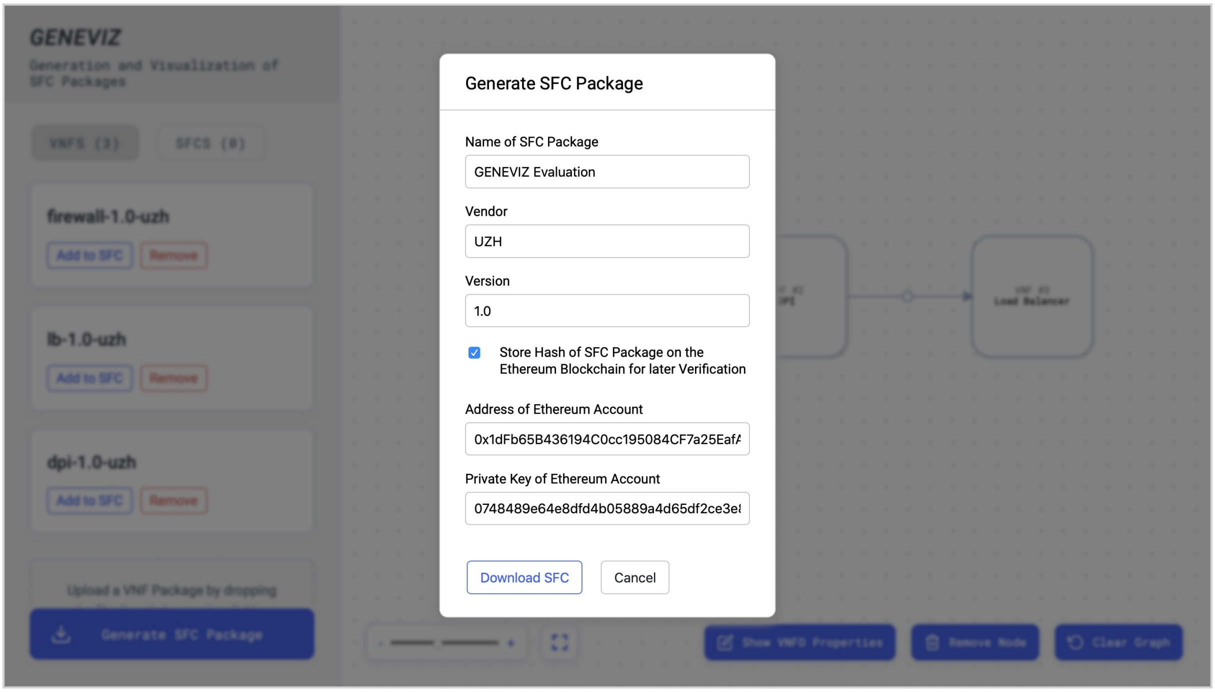


Figure 4.2: Screenshot 2 of GENEVIZ for Use Case No. 1

to "GENEVIZ Evaluation" (name), "UZH" (vendor), and "1.0" (version) respectively. The SFC Package could be downloaded at that time, but our user decides to click on the checkbox below the three input boxes in order to store the hash of the Package, which will be generated, on the Ethereum Blockchain. For this, the user needs to provide both the address of the Ethereum account as well as the private key for this account in order to sign the transaction properly. The information provided in the popup can be seen in Figure 4.3.
As a next step, the user clicks the blue-bordered "Download" button, which forces the generation of the SFC Package and the storage of the hash value of the package on the Ethereum Blockchain. The web browser should automatically trigger the download of our ZIP file, containing both an `sfc.zip` file for the deployment of the SFC as well as a `geneviz.json` file to validate the SFC Package at a later point in time.

4.0.3 Case Study No. 3 — Validation and Import of an SFC Package

For this case study, we take in a second, different user being a different actor than the one from Case Study No. 1 and 2. This second user has downloaded three SFC Packages somewhere from the Internet (e.g. from a marketplace or a website) with a specific end-user requirement in mind, having no idea about the exact content of the SFC Packages. Also, the user has no idea about who the authors of the SFC Packages were, and if one can rely on the source of the downloaded file. As the packages contain a lot of different folders and nested files — which would take quite some time to check it's content — the user leaves the validation of the packages up to GENEVIZ. The user would like to use the


Generate SFC Package

Name of SFC Package: GENEVIZ Evaluation

Vendor: UZH

Version: 1.0

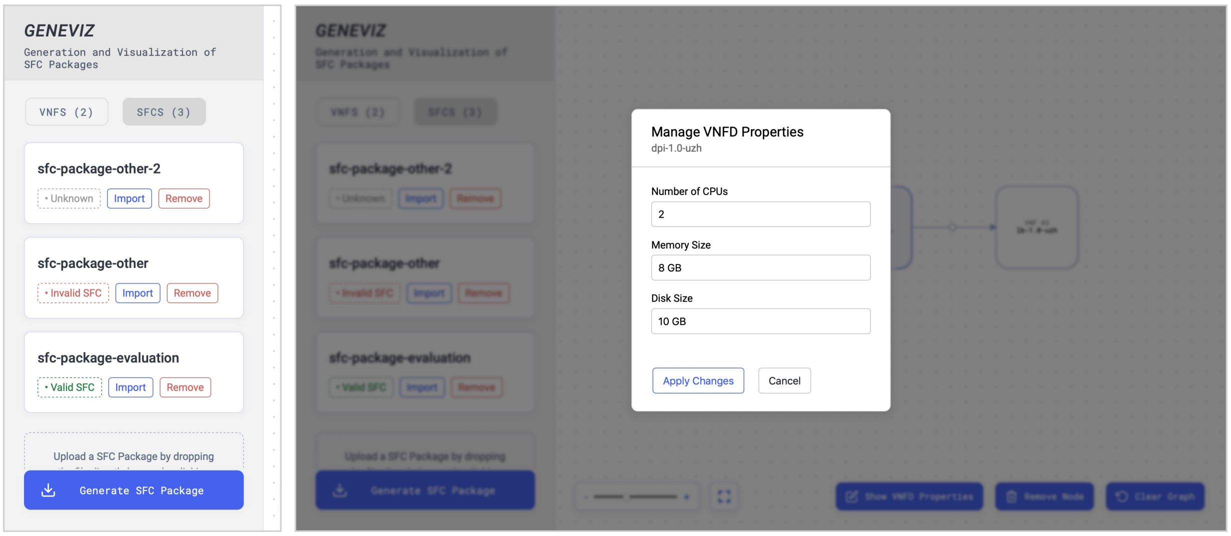
[✓] Store Hash of SFC Package on the Ethereum Blockchain for later Verification

Address of Ethereum Account: 0x1dFb65B436194C0cc195084CF7a25EafA

Private Key of Ethereum Account: 0748489e64e8dfd4b05889a4d65df2ce3e

Figure 4.3: Screenshot of GENEVIZ application for Use Case No. 2

SFC Package named `sfc-package-evaluation`, but he also considers the `sfc-package-other` and the `sfc-package-other-2` packages if the first one turns out to be invalid. In a first step, the user clicks on the grey-bordered "SFCS" button in the menu on the left side of the platform in order to switch to the SFC section. The button should now be dark colored, indicating that the "SFC" section is chosen. This section allows the manual upload of SFC Packages through the browser as it is done in the same way for the upload of VNF Packages. The uploaded SFC Packages appear now on the SFC List as *SFC Templates* in a similar way uploaded VNF Packages are handled as *VNF Templates*. The validation of the uploaded packages is triggered automatically and the response depends on the current block time of the blockchain. For the package named `sfc-package-other-2`, no statement about its data integrity can be made, as no information for the retrieval on the blockchain is provided, and it is hence marked as *Unknown*. The second package, named `sfc-package-other`, appears to be marked as *Invalid*, which means the content of the package was somehow modified. The third package named `sfc-package-evaluation` is the package we created during Case Study No. 1 and generated in Case Study No. 2. As the hash of this package was written on the Ethereum Blockchain during the generation of the package by the first user, and the transaction ID is part of the downloaded SFC Package, GENEVIZ finds a data hash for the given transaction ID, which matches with the hash of the content from the uploaded SFC Package. Hence, GENEVIZ marks the uploaded SFC Package as *Valid*, and shows a green "Valid" label at the location where the "Validate" button was before. The labels can be seen in Figure 4.4a in the white boxes of the SFC Templates.
For the last SFC Package, trust could now be established — to a certain degree — between our second user and the downloaded SFC Package from the unknown source. The user decides to import this SFC Package as a new SFC into GENEVIZ in order to adjust some VNFD properties of the involved VNFs, which fit better his demands. By clicking on the



Number of CPUs: 2
Memory Size: 8 GB
Disk Size: 10 GB

(a) Uploaded SFCs (b) VNFD Editor of the DPI VNF from the imported SFC

Figure 4.4: Screenshots of GENEVIZ application for Use Case No. 3

blue-bordered ”Import” button on the left menu, an alert appears, warning the user that the currently drafted SFC will be cleared and replaced by the new SFC. This is totally fine for our user as the previously drafted SFC is not relevant anymore and hence the ”OK” button is clicked on the alert. Now, the SFC will be imported if the SFC Package can be extracted properly and the *Service Constructor* visualization shows the corresponding graph. This is helpful for our user too, as the forwarding graph is now directly visible and could also be modified if needed.
Next, the user decides to open the *VNFD Editor* for the DPI VNF, which lies between the Firewall and the LB VNF. By clicking on the blue-filled button ”Show VNFD Properties” at the bottom of the *Service Constructor* visualization, a popup window appears, showing the current VNFD properties of the DPI VNF. Our user decides to change the memory size to ”8 GB” and the number of CPUs to ”2”, which seems to better fit the demands for the new network service. After clicking the blue-bordered ”Apply Changes” button, the new VNFD properties are directly updated.
Finally, the user clicks the blue-filled ”Generate SFC Package” button at the bottom left corner of the GENEVIZ web application, forcing the generation and download of an adjusted SFC Package based on the SFC Package created by the first user. In the same way, the information is provided for the NSD and the Blockchain in the popup window, which appears after the button is pressed, it can be done for this second SFC Package. This last step completes our evaluation and the second SFC Package could be further used to deploy a new network service.

4.0.4 Discussion

The three case studies deal with different scenarios making usage of GENEVIZ. Both the construction of a new network service based on some VNF Packages as well as the adjustment of the properties of a certain VNF — being part of the SFC — aim to be simplified with the proposed solution. With a graphical user interface for the *Service Constructor* and the *Placement Recommender*, the critical issue of VNF placement can be addressed, helping the user to draft better SFCs in terms of chaining. Further, by writing the hash of the content of the newly created package on a blockchain, the verification of the package originality can be done by a second user through GENEVIZ as well. All these tasks could be done separately, consuming unnecessary manual effort for users inexperienced in the NFV landscape, and aim to be unified and thus simplified at one single place through the GENEVIZ.

One of the limitations SFC construction currently has in general is the limitation of one single forwarding graph. Hence, GENEVIZ too only allows the definition of one output path to connect the next VNF. The case of creating multiple output paths and hence connecting multiple VNFS with a single VNF would need to creation of multiple SFCs. Further, the VNFFGD cannot be entirely generated through GENEVIZ since this depends too heavily on the NFV platform the SFC will be deployed to. Thus, the order of the VNFs is set through the order of the VNFDs in the `vnfd` property of the NSD, with the first VNF appearing in the VNFD list being the first node in the forwarding graph. In the future, there could be VNFs needing more than one output path in order to run properly. How this would be translated through one single NSD for a single SFC is not clear at the moment. It might be that GENEVIZ needs to be integrated directly into the different NFV environments (e.g. based on OpenStack Tacker [34] or CloudStack [35]) to generate the VNFFGD during the generation of a new SFC Package.

Another limitation of the work is that the evaluation is solely based on case studies. Thus, we are currently not able to provide quantitative evidence about the performance of the GENEVIZ platform in terms of intuitiveness and simplification to create SFCs *easier* and *faster* than in a manual way. In this sense, a usability evaluation with real users could be done to validate the real benefits of GENEVIZ and provide details about the effectiveness of the platform. An evaluation using the System Usability Scale (SUS) questionnaire and other methods are mapped as future work.

We take into account the usage of a blockchain to ensure the data integrity of a previously created SFC Package. First, for a smooth user experience, we assume the block time to be in a reasonable amount of time for the end user of GENEVIZ, which is not always guaranteed for certain blockchains (e.g. Ethereum had several peaks with up to 30 seconds for the block time in its recent history [36]). Second, although the integrity of the content can be guaranteed through the hash verification, the package could already contain malicious code at the moment of the creation of the package, inserted by the creator of the SFC Package. Or malicious code could be inserted even earlier during the creation of a VNF Package, which would, in turn, be part of the downloaded SFC Package. Hence, actually ensuring *trust* on a downloaded package from the internet can not be guaranteed with GENEVIZ and the user also needs to trust the creator of the package itself. Checking the content of the package would still be up to the marketplace provider or the catalog holder when approving a new package on their respective platforms, or even the second user from Case Study No. 2, who finally validates the downloaded package and wants to use them further to create a new network service.

Chapter 5

Future Work

Due to the modularized nature of GENEVIZ, several different extensions could be explored as future research directions. Besides the further enhancement of existing modules inside GENEVIZ, we could imagine the expansion of the platform with additional features for the internal components or even the coupling with external components.

First, the *Placement Recommender* visualization could be expanded in order to support further ways to recommend the placement of certain VNFs inside the service function chaining. For the GENEVIZ Prototype, we considered the addition of two new properties inside the VNFD to be able to make a statement about the chaining itself. Hence, the creator of a VNF Package would be responsible for adding the necessary information with these two properties. We are aware that this can not always be guaranteed. As a future work, we could imagine an enhanced *Placement Recommender*, taking information from a publicly accessible database in order to make a statement about the chaining. Since this information would be based on large datasets of productive NFV environments, we could also envisage the application of machine learning on these datasets, making it even possible to define certain performance requirements in GENEVIZ at the beginning of the creation of a new network service, which would therefore serve as a base for the placement recommendation not only in terms of chaining, but also in terms of performance. By considering the work from Jacobs *et al.* [37], the affinity between pairs of VNFs, which is based on a weighted set of criteria, could be computed with the help of past empirical data and be presented in the *Placement Recommender* visualization (e.g. a number in the range of [0,1] is shown a label on the edge, with 1 being fully recommended and 0 not recommended under any circumstances). Further, the recommender is in general not limited to the two new properties from the VNFD we proposed in Chapter 3. Based on information provided by Open Baton [38], the NSD can contain an optional property named `vnf_dependency`, which describes the dependency of the target VNF from a VNFD property of the source VNF. Hence, when considering these dependencies as well, a certain order of the VNFs would even be mandatory during the service construction and should also be provided by an enhanced *Placement Recommender*.

Secondly, the *VNFD Editor* could be made configurable to support the editing of more properties from the VNF Descriptor. This could easily be managed by the current GENEVIZ Prototype as well, as the input fields in the popup window map directly to the properties of the VNFD. Further, the *VNFD Editor* could also be extended in a broader sense to a so-called *VNF Editor*. It would perhaps make sense to adjust more than only the properties of the VNFD and the editing could also include the adjustment of launch or management configuration files.

Thirdly, for the GENEVIZ Prototype, we require the VNF Packages to be in a ZIP file format for the manual upload. As these files need to be created beforehand, in a future version of GENEVIZ, we would like to be able to create new VNF Packages through the same web application as it can be done for SFC Packages. Hence, GENEVIZ would also allow the creation of completely new VNF Packages from scratch and the import and adjustment of previously created VNF Packages. The latter goes in a similar direction to the one of the extended *VNFD Editor*, but a *VNF Constructor* would also allow the upload of completely new files instead of the editing of properties in already existing files. These newly created VNF Packages could then be used on-the-fly for a new SFC Package. In the same way, the hash of newly created SFC Packages is written on a blockchain, it could be done for newly created VNF Packages, addressing the problem of malicious code and data integrity on them as well.

Lastly, the *SFC Validator* could be improved not only to ensure data integrity but also to additionally increase the level of trust for an SFC Package from an unknown source by scanning the content for malicious content in general. This could be done both during the construction of an SFC, while VNF Packages are imported, as well as during the import of an existing SFC Package.

Chapter 6

Summary and Conclusions

As NFV becomes more popular, technically more mature, and its infrastructure wider adopted, the demand for specific network services based on the chaining of different virtualized network functions could increase significantly in the years to come. In this thesis, we introduced GENEVIZ, a tool for the generation, validation, and visualization of SFC Packages. We propose that a graphical user interface can lead to a *more intuitive* and *easier* construction of new network services. GENEVIZ could also lead to fewer mistakes during the crafting of new, or the adjustment of existing services, as there are fewer steps necessary to generate an SFC Package. In this thesis, we presented and discussed different visualizations, brought together into one single web application: i) the construction of an SFC by chaining different VNFs through a single, directed, and acyclic graph, ii) the adjustment of VNFD properties of the VNFs being part of the SFC, iii) supporting the user to create better SFCs in terms of chaining by providing a placement recommender, and iv) giving the ability to validate a previously created SFC to check its data integrity.

After having set out the introduction and motivation of GENEVIZ, some theoretical background is given on NFV, SFCs, and blockchain technology in general. With this background in mind, related work which applies information visualization in the context of NFV and SFC is discussed. In a next section, the conceptual architecture of GENEVIZ is presented, and, after outlining some overall use cases for GENEVIZ, each component is explained in a more detailed manner. Further, an evaluation of the GENEVIZ Prototype is conducted based on three different use case scenarios and followed by a discussion of the prototype. Possible extensions of GENEVIZ, fitting into the context of SFC creation and adjustment, are then set out in the *Future Works* section.

GENEVIZ aims to be a web application running on a local machine, hence allowing end users to create and adjust SFC Packages locally, which, in turn, removes the need of trust into any provider of GENEVIZ. Although the tool provides the possibility to validate the data integrity of the content of a package by using the decentralized blockchain technology, trust in the integrated VNF Packages is still necessary in the end. The modularity of GENEVIZ offers wide extensibility and we could imagine a whole bunch of appendices into the platform, enabling an even richer application, and, thus, supporting the construction of new services even better, while the NFV infrastructure finds widespread dissemination. A tool, such as GENEVIZ, finally has the potential not only to help experienced network operators but also complete newbies, furthermore driving the wide adoption of NFV technology. However, the effectiveness and usability of GENEVIZ still need to be proven by considering real use case scenarios with real end users.

Bibliography

[1] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys Tutorials*, 18(1):236–262, Firstquarter 2016.

[2] Chiosi, Margaret and Clarke, Don and Willis, Peter and Reid, Andy and Feger, James and Bugenhagen, Michael and Khan, Waqar and Fargano, Michael and Cui, Chunfeng and Deng, Hui and others. Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action. In *SDN and OpenFlow World Congress*, volume 48. sn, 2012.

[3] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, Feb 2015.

[4] L. Bondan, M. F. Franco, L. Marcuzzo, G. Venancio, R. L. Santos, R. J. Pfitscher, E. J. Scheid, B. Stiller, F. De Turck, E. P. Duarte, A. E. Schaeffer-Filho, C. R. P. d. Santos, and L. Z. Granville. Fende: Marketplace-based distribution, execution, and life cycle management of vnfs. *IEEE Communications Magazine*, 57(1):13–19, January 2019.

[5] V. T. Guimarães, C. M. D. S. Freitas, R. Sadre, L. M. R. Tarouco, and L. Z. Granville. A survey on information visualization for network and service management. *IEEE Communications Surveys Tutorials*, 18(1):285–323, Firstquarter 2016.

[6] M. F. Franco, R. L. d. Santos, A. Schaeffer-Filho, and L. Z. Granville. Vision – interactive and selective visualization for management of nfv-enabled networks. In *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, pages 274–281, March 2016.

[7] ETSI NFV ISG. Network Functions Virtualisation, June 2012. White Paper. Accessed on November, 2018.

[8] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *2010 Proceedings IEEE INFOCOM*, pages 1–9, March 2010.

[9] ETSI GS NFV-MAN. Network Functions Virtualisation (NFV); Management and Orchestration . Dez 2014.

[10] J. Zhang, Z. Wang, N. Ma, T. Huang, and Y. Liu. Enabling efficient service function chaining by integrating nfv and sdn: Architecture, challenges and opportunities. *IEEE Network*, 32(6):152–159, November 2018.

[11] J. A. Wickboldt, W. P. De Jesus, P. H. Isolani, C. B. Both, J. Rochol, and L. Z. Granville. Software-defined networking: management requirements and challenges. *IEEE Communications Magazine*, 53(1):278–285, January 2015.

[12] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck. Network slicing and softwarization: A survey on principles, enabling technologies, and solutions. *IEEE Communications Surveys Tutorials*, 20(3):2429–2453, thirdquarter 2018.

[13] S. Van Rossem, W. Tavernier, B. Sonkoly, D. Colle, J. Czentye, M. Pickavet, and P. Demeester. Deploying elastic routing capability in an sdn/nfv-enabled environment. In *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pages 22–24, Nov 2015.

[14] Nakamoto, Satoshi and others. Bitcoin: A peer-to-peer electronic cash system. 2008.

[15] T. Aste, P. Tasca, and T. Di Matteo. Blockchain technologies: The foreseeable impact on society and industry. *Computer*, 50(9):18–28, 2017.

[16] Popov, S. The Tangle, IOTA Whitepaper, 2018.

[17] Ware, Colin. *Information visualization: perception for design.* Elsevier, 2012.

[18] L. R. Soles, T. Reichherzer, and D. H. Snider. A tool set for managing virtual network configurations. In *SoutheastCon 2016*, pages 1–4, March 2016.

[19] I. J. Sanz, D. M. F. Mattos, and O. C. M. B. Duarte. Sfcperf: An automatic performance evaluation framework for service function chaining. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9, April 2018.

[20] R. A. Eichelberger, T. Ferreto, S. Tandel, and P. A. P. R. Duarte. Sfc path tracer: A troubleshooting tool for service function chaining. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 568–571, May 2017.

[21] Timo Hegnauer. Design and Development of a Blockchain Interoperability API. Master's thesis, CSG@IFI, University of Zurich, Switzerland, to appear 2019. Supervisors: Eder Scheid, Bruno Rodrigues, Burkhard Stiller.

[22] Facebook Inc. React — A JavaScript Library for Building User Interfaces. `https://reactjs.org`, accessed 21 April, 2019.

[23] Fedosejev, Artemij. *React. js Essentials.* Packt Publishing Ltd, 2015.

[24] Dan Abramov and the Redux documentation authors. Redux — A Predictable State Container for JS Apps. `https://redux.js.org`, accessed 21 April, 2019.

[25] Microsoft Corporation. TypeScript — JavaScript that scales. `https://www.typescriptlang.org`, accessed 21 April, 2019.

[26] npm, Inc. NPM — The Essential JavaScript Development Tool. `https://www.npmjs.com`, accessed 21 April, 2019.

[27] Sebastian McKenzie and other contributors. Babel — The Compiler for Next Generation JavaScript. `https://babeljs.io`, accessed 21 April, 2019.

[28] JS Foundation and other contributors. webpack Website. `https://webpack.js.org`, accessed 21 April, 2019.

[29] Python Software Foundation. The Python Programming Language. `https://www.python.org`, accessed 21 April, 2019.

[30] Armin Ronacher. Flask (A Python Microframework). `http://flask.pocoo.org`, accessed 21 April, 2019.

[31] Wood, Gavin and others. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

[32] Truffle Blockchain Group. Ganache Website. `https://truffleframework.com/ganache`, accessed 21 April, 2019.

[33] Piper Merriam and Jason Carver. Web3.py Documentation. `https://web3py.readthedocs.io/en/stable/`, accessed 21 April, 2019.

[34] OpenStack Foundation. OpenStack Docs — Tacker Documentation. `https://docs.openstack.org/tacker/latest/`, accessed 21 April, 2019.

[35] Apache Software Foundation. Apache CloudStack. `https://cloudstack.apache.org`, accessed 21 April, 2019.

[36] D. Vujičić, D. Jagodić, and S. Ranđić. Blockchain technology, bitcoin, and ethereum: A brief overview. In *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pages 1–6, March 2018.

[37] A. S. Jacobs, R. L. do Santos, M. F. Franco, E. J. Scheid, R. J. Pfitscher, and L. Z. Granville. Affinity measurement for nfv-enabled networks: A criteria-based approach. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 125–133, May 2017.

[38] Open Baton. OpenBaton Documentation — Network Service Descriptor. `https://openbaton-docs.readthedocs.io/en/2.1.3/ns-descriptor/`, accessed 21 April, 2019.

[39] Martin Juan José Bucher. GENEVIZ Prototype — Source Code. `https://github.com/mnbucher/geneviz`, accessed 21 April, 2019.

[40] Martin Juan José Bucher. GENEVIZ Prototype — Source Code. `https://gitlab.ifi.uzh.ch/franco/geneviz`, accessed 21 April, 2019.

Abbreviations

AI	Artifical Intelligence
API	Application Programming Interface
BLOB	Binary Large Object
CAPEX	Capital Expenditures
CDN	Content Delivery Network
CLI	Command-Line Interface
CSG	Communication Systems Research Group
DAG	Directed Acyclig Graph
DPI	Deep Packet Inspection
IP	Internet Protocol
LB	Load Balancer
MVC	Model-View-Controller
NAT	Network Address Translation
NF	Network Function
NFV	Network Function Virtualization
NFVI	Network Function Virtualization Infrastructure
NFVO	Network Function Virtualization Orchestrator
NPM	Node Package Manager
NSD	Network Service Descriptor

List of Figures

List of Tables

Appendix A

HTTP API of the Data Layer

By using Flask, the following six HTTP Endpoints are provided in order to establish the communication between the *User Layer* and the *Data Layer* of GENEVIZ. The list of parameters and response types should serve as additional help besides the source code of the GENEVIZ prototype:

- Store VNF Template: Table A.1
- Get VNFD: Table A.2
- Update VNFD: Table A.3
- Import SFC Package: Table A.4
- Validate SFC Package: Table A.5
- Generate SFC Package: Table A.6

Table A.1: Request to store VNF Package

Request		
Path	HTTP verb	Description
`/vnfs`	POST	Stores the .zip of the VNF Template in a temporary folder on the local machine.
Body Parameters for the Request		
Name	Data Type	Description
`uuid`	string	UUID of the VNF Template (given by the User Interface)
`vnfName`	string	Name of the VNF Template
`fileBase64`	string	Content of the .zip file encoded in Base64 format and converted to a string
Response with HTTP Status 200		
Name	Data Type	Description
`success: True`	boolean	VNF Package could be stored successfully
Response with HTTP Status 400		
Name	Data Type	Description
`success: False`	boolean	VNF Package could not be stored successfully

Table A.2: Request to get VNF Descriptor

Request		
Path	HTTP verb	Description
`/vnfs/<vnf_name>/<uuid>`	GET	Extracts the VNFD from the VNF Package .zip file and returns it as a JSON object
URL Parameters for the Request		
Name	Data Type	Description
`vnf_name`	string	Name of the VNF Package (should refer to the name of the VNF Template)
`uuid`	string	UUID of the VNF Package
Response with HTTP Status 200		
Name	Data Type	Description
`success: True`	boolean	VNFD could be retrieved successfully
`vnfd`	JSON object	VNFD of a VNF Package in JSON format
Response with HTTP Status 400		
Name	Data Type	Description
`success: False`	boolean	VNFD could not be retrieved

Table A.3: Request to update VNF Descriptor

Request		
Path	HTTP verb	Description
`/vnfs/<vnf_name>/<uuid>`	PUT	Updates the VNFD of a given VNF Package which is already stored
URL Parameters for the Request		
Name	Data Type	Description
`vnf_name`	string	Name of the VNF Package (should refer to the name of the VNF Template)
`uuid`	string	UUID of the VNF Package
Body for the Request		
Name	Data Type	Description
-	string	Serialized `JSON` of the VNFD Package
Response with HTTP Status 200		
Name	Data Type	Description
`success: True`	boolean	VNFD could be udpated successfully
Response with HTTP Status 400		
Name	Data Type	Description
`success: False`	boolean	VNFD could not be updated

Table A.4: Request to import SFC Package

Request		
Path	HTTP verb	Description
`/sfcs`	POST	Updates the VNFD of a given VNF Package which is already stored
Body for the Request		
Name	Data Type	Description
-	string	Serialized `string` of the encoded .zip file of the SFC Package in Base64 format
Response with HTTP Status 200		
Name	Data Type	Description
`success: True`	boolean	VNFD could be udpated successfully
`vnfs`	object[]	Array of VNF Packages with each element containing three attributes, namely `name` for the name of the VNF Template, `uuid` referring to the VNF Package, and `vnfd` with the corresponding VNFD of the VNF Package
`order`	string[]	Array of UUIDs as strings, which define the order of the VNF Packages for the graph construction
Response with HTTP Status 400		
Name	Data Type	Description
`success: False`	boolean	VNFD could not be updated

Table A.5: Request to validate SFC Package

Request		
Path	HTTP verb	Description
`/sfcs/validate`	POST	Updates the VNFD of a given VNF Package which is already stored
Body for the Request		
Name	Data Type	Description
-	string	Serialized `string` of the encoded .zip file of the SFC Package in Base64 format
Response with HTTP Status 200		
Name	Data Type	Description
`success: True`	boolean	SFC Package is valid
Response with HTTP Status 404		
Name	Data Type	Description
`success: False`	boolean	SFC Package is not valid
Response with HTTP Status 400		
Name	Data Type	Description
`success: False`	boolean	SFC Package is unknown. Either no information could be found on the Blockchain or the SFC Package has a wrong folder structure such that the validation could not be done properly.

Table A.6: Request to generate SFC Package

Request		
Path	HTTP verb	Description
`/sfcs/generate`	POST	Updates the VNFD of a given VNF Package which is already stored
Body Parameters for the Request		
Name	Data Type	Description
`vnfPackages`	string	Name of the VNF Package (should refer to the name of the VNF Template)
`path`	string	UUID of the VNF Package
`nsd`	object	Object containing three attributes, namely `name` (string), `vendor` (string), and `version` (string) for the Network Service Descriptor
`bc`	object	Object containing three string attributes, namely `storeOnBC` (boolean), `address` (string), and `privateKey` (string). If the Hash should not be stored on the Blockchain, `storeOnBC` will be false and the remaining two attributes should contain an empty string. If the Hash should be stored, the boolean will be true and the two string attributes should contain the necessary data to store the Hash on the Blockchain.
Response with HTTP Status 200		
Name	Data Type	Description
`-`	BLOB	SFC Package could be created successfully
Response with HTTP Status 400		
Name	Data Type	Description
`success: False`	boolean	SFC Package could not be created

Appendix B

Installation Guidelines

This chapter provides the necessary information to install and run the components of the GENEVIZ Prototype on a computer or virtual machine with a fresh installation of Apple's macOS. Setting up the components on another operating system based on UNIX should work quite similar, for Windows the steps could differ slightly more. The source code of the GENEVIZ Prototype is available at Github [39] or the Gitlab [40] of the Institute of Informatics at the University of Zurich.

B.1 Setting up the User Layer

Setting up the User Layer of the GENEVIZ Prototype requires first the installation of the Node Package Manager (npm). The distribution of *npm* comes along with *Node.js* and can be installed from the website:

```
https://www.npmjs.com/get-npm
```

After navigating into the root directory of the source code, all necessary node packages defined in the *packages.json* file need to be installed by running the following command through the Command-Line Interface (CLI):

```
npm install
```

This should have successfully installed all node modules in the *node_modules* directory. As the source code of the web application is written in TypeScript, the language of the package named *react-digraph* needs to be manually changed to TypeScript instead of regular JavaScript by opening the following file in order to edit it:

```
node_modules/react-digraph/package.json
```

Now, the `package.json` file needs to appended by adding a new attribute called `typings` in the first hierarchy-level of the JSON in order to refer to the following path:

```
"typings": "./typings/index.d.ts"
```

After saving the edited *package.json* of the `react-digraph` module, the web application should now be ready to be properly compiled in development mode. The development mode comes with an integrated development server on *localhost*, which enables live reloading if the source code has changed. In favor of this, not all source code files are fully optimized. To start the development mode, run the following command through the CLI, which also shows up the respective port for the web application:

```
npm run dev
```

To compile the application in production mode, run the following command:

```
npm run build
```

B.2 Setting up the Data Layer

In order to run the *Data Layer* of GENEVIZ, Python 3.7 needs to be installed first, which can be done through the Website:

```
https://www.python.org/downloads/
```

After successfully installing Python 3.7, the global installation of the `virtualenv` package needs to be installed with the help of the Python Package Installer (pip). Pip should already be installed with the Python 3.7 distribution and can be accessed through the CLI by using `pip`. Sometimes, the `pip3` command is necessary, in case that `pip` is not recognized as a command. The `virtualenv` package is installed as follows:

```
pip install virtualenv
```

Now, one needs to navigate into the subfolder *geneviz-management-api*, where the source code for the *Data Layer* is located. We have put all the necessary source code for the Data Layer inside the Management API. There, a new virtual environment needs to be set up with the following command:

```
python3 -m venv venv
```

This should have created a virtual environment named `venv`. The folder name could also be named differently by changing the last argument of the above command. Now, the virtual environment needs to be activated:

```
source venv/bin/activate
```

At this step, it should be ensured that the environment is using Python 3.7 and pip3 by checking their respective versions with the `-version` flag through the CLI. As a next step, the required Python Packages need to be installed by running the following command:

```
pip install -r requirements.txt
```

Next, the `FLASK_APP` environment variable needs to be set on the local machine based on the absolute path of the `geneviz_management_api.py` file:

```
export FLASK_APP=absolute/path/to/geneviz_management_api.py
```

Lastly, still being in the virtual environment, Flask can finally be started with the following command:

```
flask run
```

This should have started Flask on the port given in the console log.

Due to access control checks, depending on the locations of both the web application, as well as the *Management API*, the respective origins of the web application need to be defined in the *Management API*. If `npm run dev` is called for the web application, running on port 3000, and the *Management API* is running on the same local machine, the communication should work properly.

If these settings differ slightly (e.g. another port or by putting the *Management API* on a remote server), the respective origin of the web application may need to be set in the *Management API* by appending or changing the `"origins"` array on line 24 of the `geneviz_management_api.py` file:

```
CORS(app, resources={r"/*": {"origins": [
    "http://localhost:3000"
]}})
```

B.3 Setting up Ganache

To set up Ganache on the local machine, it can simply be installed through the Website:

```
https://truffleframework.com/ganache
```

By clicking ”Quickstart” after the launch of the application, a local Ethereum Blockchain is created, which can then be used for testing purposes. The respective port of Ganache can be adjusted and is seen in the application window at the top menu bar. If the *Ethereum API* of the GENEVIZ Prototype can't reach Ganache, the port maybe needs to be changed in the source code of the `ethereum_api.py` file.

Appendix C

Contents of the CD

- GENEVIZ Prototype (Source Code)
- *LaTeX* Source Code of the Bachelor Thesis Document
- Bachelor Thesis Document (.pdf)
- Intermediate Presentation (.pdf)
- End Presentation (.pdf)
- Abstract German (.txt)
- Abstract English (.txt)