Informatica — Universiteit van Amsterdam

Bachelor Informatica

University of Amsterdam

Extending a simulation framework for virtualised networking infrastructures to leverage the potential of machine learning

Tom J. Wassing

September 6, 2021

Supervisor(s): Dr. Chrysa Papagianni

Signed:

Abstract

In the modern cloud, the combination of virtualisation and Software-Defined Networking (SDN) forms the basis for the current cloud architecture. The advances in virtualisation led to a cost-effective way of providing network functions by virtualising them in the cloud and removing the need for expensive proprietary hardware. SDN's dynamic and programmable properties provide centralised resource management, leading to more flexibility and faster network adaptation. Combining these two principles makes it possible to completely virtualise and host networks within a virtualised environment, allowing them to form services by chaining virtualised network functions, referred to as Service Function Chaining (SFC). Allocating resources to SFCs in an optimal way to reduce idle resources and energy waste is called the SFC embedding problem. The resource allocation problem has been approached using traditional optimisation methods and heuristics. In recent research, machine learning approaches have emerged with promising results. The CVI-Sim simulator has been developed in Java to evaluate and benchmark various resource allocation algorithms; however, Java lacks a vibrant machine learning ecosystem. Thus, in the context of this thesis, the CVI-Sim simulator has been extended to allow for integration with other programming languages, providing access to richer machine learning ecosystems. The extension introduces a new interface to external programs using an asynchronous messaging system, decoupling the admission control process from the simulator. In the evaluation, the proposed approach is compared to a reference implementation. A slight increase in the run time is measured due to the overhead introduced by the new communication layer, imposing a longer simulation time. The results of the experiments validate the workings of the extended simulator and show that it can now be used to develop and evaluate ML-based resource allocation algorithms exploiting readily available open-source libraries for AI/ML. As a result, researchers can focus on designing and evaluating new resource allocation algorithms rather than spending time on developing the corresponding ML tools or using unsupported Java implementations.

Contents

CHAPTER 1

Introduction

The internet has seen enormous growth in the last decades [33, 20] and is growing steadily [30, 20]. The growth has led to increasing amounts of internet traffic, which has resulted in many advances in networking technology, providing cloud operators with new ways to manage their data centre resources efficiently.

In today's internet architecture, *Software-Defined Networking* (SDN) and virtualisation form the basis for the modern cloud. Combining the two principles makes it possible to build a network in a virtualised environment without the need for specialised physical hardware. Virtualisation provides a lot of flexibility and scalability for networks. *Virtualised Network Functions* (VNFs) can be used to form a service by using the capabilities of SDN to create a chain of VNFs. The chaining of virtualised network functions is referred to as *Service Function Chaining* (SFC).

On average, 85% of the resources in distributed systems remain idle [4]. This indicates that there are opportunities for cloud providers to reduce energy waste and cost. Managing resources efficiently boils down to allocating computational, and transport resources to service function chains optimised regarding utilisation and cost. However, the optimisation of resources is limited by the constraints of the VNFs to deliver the service requirements. The resource allocation problem is often referred to as *SFC embedding problem.* The SFC embedding problem is related to the *Virtual Network Embedding* (VNE) problem [12] and can be seen as an extension of the knapsack problem [45], with a minimisation objective and additional constrains to preserve the workings of the VNFs. Numerous approaches have addressed the resource allocation problem as a traditional optimisation problem [35] using techniques such as numerical optimisation and heuristics.

Artificial intelligence and machine learning have recently been used to solve complex problems, such as playing a game of Go [38] or autonomous driving [9]. Reinforcement learning is most often used to solve these kinds of complex problems, in which the agent improves itself based on direct feedback without supervision, simulating human-like trial-and-error learning [29]. More recently, reinforcement learning has been used as a new approach to the SFC embedding problem because of the dynamic nature of network environments and the computational complexity of existing approaches. In particular, the (Deep) Q-learning algorithm has been suggested and showed positive results [28, 10, 7].

The Java-based simulator called *CVI-Sim* has been developed in the context of the FP7 project Networking Innovations over Virtualised Infrastructures (NOVI) Papagianni et al. [34] and has been extended accordingly in the following years [35] to serve as a simulation environment for resource allocation approaches applied on virtualised infrastructures, from data centres to 4G networks. The Python programming language is often used in the machine learning community due to its high-level nature allowing for quick prototyping. This has led to a vibrant machine learning ecosystem with many actively developed libraries containing cutting-edge research and optimisations. Java has a much smaller machine learning ecosystem; researchers using CVI-Sim, therefore, have to be more self-reliant and often have to reinvent the wheel. Which can lead to longer development cycles with the potential for more bugs. This research will accommodate the issue, leading to the following research question:

"How to extend the CVI-Sim with an interface to provide for a loosely coupled integration with external machine learning ecosystems, to allow for experimentation of machine learning approaches to tackle the SFC embedding problem¿

1.1 Ethical considerations

In 2018, data centres consumed approximately 200 terawatt-hours (TWh), which is 1% of the global electricity use [20]. It has also been predicted that the overall energy consumption will grow steadily with the inevitable increase of IT usage [30]. It has been shown that data centres contain a lot of idle resources [4]. Reducing these idle resources by using resource allocation techniques is not only crucial for data centre operators to reduce costs and increase profit margins, but it also contributes to the sustainability of data centres. By doing so, the industry can advance while minimising its impact on the environment, with the ultimate goal of reversing global warming.

1.2 Outline

The thesis is organised as follows. In Chapter 2 background information is provided, which gives a basic understanding of the SFC embedding problem and how it is currently approached. Furthermore, the inner workings of the CVI-Sim simulator are discussed and is compared to other popular simulators. At last, reinforcement learning is reviewed and how it is currently applied to the SFC embedding problem. Chapter 3 presents the design of the CVI-Sim extension and outlines the design decisions that have been made. Describing the new architecture and requirements of each component. The implementation of the design is described in Chapter 4, explaining how the components have been implemented and which libraries have been used. The extension is validated by replicating the reinforcement learning approach to the SFC embedding problem from the research of De Vleeschauwer et al. [7]. The replication serves as the test implementation and is described in Chapter 5. In Chapter 6 the evaluation method, experiments, and criteria are stated. Outlining how the test implementation is compared to the reference implementation. At last, the results of the experiments are presented. Finally, in Chapter 7, the results are discussed, the research question is concluded, and further research is suggested.

CHAPTER 2

Background

This chapter first explains the SFC embedding problem and how it is currently approached. In the following, existing simulation environments for testing embedding algorithms are compared and discussed in contrast to CVI-Sim. In addition, the inner works of CVI-Sim are further investigated, revealing how the simulations are currently done and how the project is structured. Finally, the reinforcement learning algorithm Q-learning is explained, serving as a basis for the extension's design and test implementation.

2.1 SFC embedding problem

Service function chaining, also known as network service chaining, is a functionality that uses SDN capabilities to connect a set of virtualised functions forming essentially a chain, for the purpose of delivering a network service. The virtualised network functions are mapped to the underlying substrate network's physical resources, as shown in the Figure 2.1. Allocating physical resources (*e.g.,* CPU, bandwidth, etc.) to the service function chain in an optimal way is known as the SFC embedding problem.

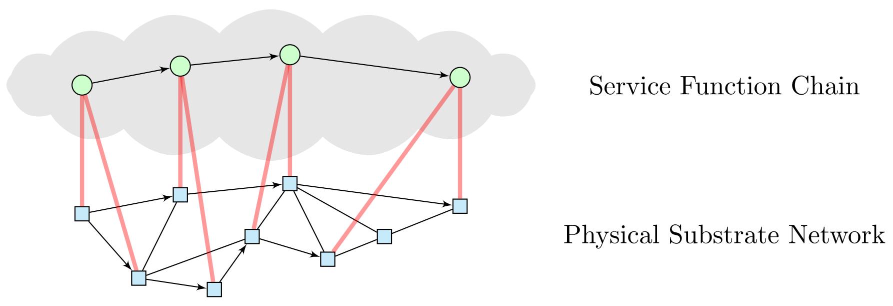


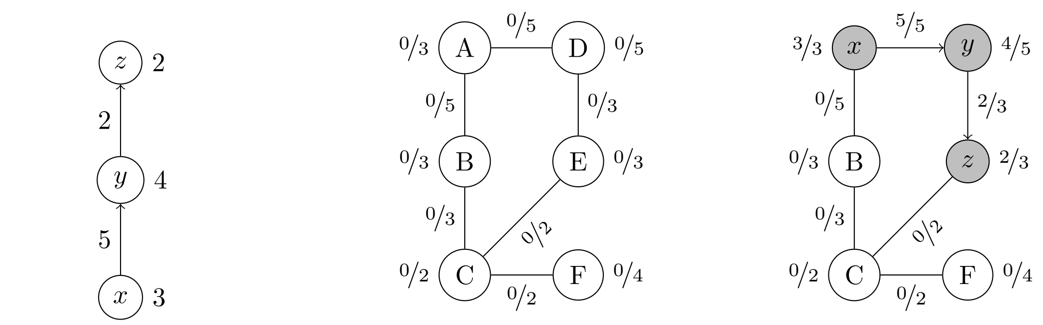
Figure 2.1: Two possible mappings of a service function chain to the substrate network resources

SFC embedding can be divided into three stages [12]. The first stage is called *VNFs-Chain Composition* (VNFs-CC), also known as SFC. During the VNFs-CC stage, Virtual Network Functions Requests (VNFR) are composed into a chain, resulting in a forward-directed acyclic graph. The graph contains the service requirements (*e.g.,* bandwidth, computing capacity) that have to be met for the chain to function correctly. In the second stage, called *VNF-Forwarding Graph Embedding* (VNF-FGE), the VNFs in the forward graphs are mapped to the resources in the underlying substrate network. This phase's challenge is where to allocate the VNFs in the most optimal way to meet the service requirements and maximise efficiency. The third and final stage is the *VNF-Scheduling* (VNFs-SCH) stage. During the VNFs-SCH stage, the network functions are scheduled to minimise the total execution time without exceeding the Service Level Agreement (SLA).

The SFC embedding problem is essentially the challenge of the VNF-FGE stage. The allocation problem is known to be $\mathcal{NP}$-hard [1] since it is a generalisation of the VNE problem [12]. The problem is most often approached using (Mixed) Integer Programming (MIP/IP) or greedy allocation algorithms. Due to the complexity of the problem, most research focuses on *relaxing* the problem by designing heuristics and meta-heuristic algorithms [12], making the computation less taxing.

Example

Figure 2.2 illustrates an example SFC embedding scenario, highlighting the first two stages. In the scenario, a service request has been composed into a graph, shown in Figure 2.2a. The graph's nodes represent the VNFs, denoted as x, y, and z. The nodes and links are annotated with their required computing and bandwidth capacities. In the example, the graph is embedded on an idle substrate network displayed in Figure 2.2b, each node and link indicated with their available capacity. A possible embedding of the service function chain is demonstrated in Figure 2.2c.



(a) SFC forward graph (b) Substrate network (c) SFC embedding

Figure 2.2: SFC embedding example scenario

2.1.1 Formulation

The SFC embedding problem can be formulated in MIP as in Papagianni, Papadimitriou, and Baras [35]. The substrate network is represented as a directed weighted graph $G_S = (N_S, E_S)$, where N_S represents the set of all nodes (*i.e.,* routers, switches, and servers) and E_S the corresponding links of the substrate network. The set N_v represents the set of all possible virtualised network functions (*e.g.,* firewall, DPI, NAT, etc.) that can be deployed on the nodes of the substrate network G_S. Each VNF is associated with a computing demand g^i. The service function chain request can be represented as a directed weighted acyclic graph $G_F = (N_F, E_F)$, where $N_F \subseteq N_v$. The edges are expressed as $(i, j) \in E_F$, associated with their bandwidth demands defined as g^{ij}.

The placement of the VNFs $i \in N_v$, in the service chain request, on the substrate network nodes $u \in N_a$ can be expressed as the binary variable x_u^i. The real variable $f_{uv}^{i,j}$ dictates the amount of bandwidth assigned to the link between $(u, v) \in E_S$, used by the edge $(i, j) \in E_F$. The maximum computational capacity of the nodes in the substrate network is limited to r_u, and the available bandwidth per link is defined by r_{uv}.

Objective:

$$\text{Min.} \sum_{i \in N_F} \sum_{u \in N_S} x_u^i + \frac{1}{\sum\limits_{(i,j) \in E_F} g^{ij}} \sum_{(i,j) \in E_F} \sum_{\substack{(u,v) \in E_S \\ (u \neq v)}} f_{uv}^{ij} \tag{2.1}$$

Capacity related constraints:

$$\sum_{\forall i \in N_F} g^i x_u^i \leq r_u \quad \forall u \in N_S \tag{2.2}$$

$$\sum_{\forall (i,j) \in E_F} f_{uv}^{ij} \leq r_{uv} \quad \forall (u,v) \in E_S \tag{2.3}$$

Placement related constraint:

$$\sum_{\forall u \in N_S^x} x_u^i = 1 \quad \forall i \in N_F \tag{2.4}$$

Flow related constraint:

$$\sum_{\substack{v \in N_S \\ (u \neq v)}} (f_{uv}^{ij} - f_{vu}^{ij}) = g^{ij}(x_u^i - x_u^j) \quad i \neq j, \forall (i,j) \in E_F, u \in N_S \tag{2.5}$$

Domain constraints:

$$x_u^i \in \{0,1\} \quad \forall i \in N_F, u \in N_S \tag{2.6}$$

$$f_{uv}^{ij} \geq 0 \quad \forall (u,v) \in E_S, (i,j) \in E_F \tag{2.7}$$

The objective of the formulated SFC embedding problem is to minimise the objective function 2.1. The objective function is minimised over the variable f, to reduce link usage on the substrate network. The first term of the objective function represents the total amount of allocated computing resources, which is fixed. The second term represents the total amount of allocated bandwidth. The capacity related constraints ensure that the used computational and bandwidth resources are within the bounds of the available capacity of the substrate network. Constraint 2.2 limits the total amount of computational capacity, and constraint 2.3 limits the total amount of bandwidth. The placement constraint 2.4 ensures that a VNF is allocated at most once to a node in the underlying substrate network. Constraint 2.5 ensures flow conservation of the links. At last, constraints 2.6 and 2.7 expresses the domain of the variables x_u^i and f respectively.

2.2 Simulation environments

Testing and evaluating algorithms to tackle the resource allocation problem is difficult and expensive to do with actual equipment at scale. Simulators are often used for prototyping and benchmarking new algorithms. The simulator can emulate different topologies with various traffic patterns, allowing researchers to test the algorithm from different perspectives. Some simulators use real-world anonymised internet traffic to simulate real-world scenarios [28]. Numerous simulators have been developed, each with different goals [39]. In the following sections, various simulators are described and compared to CVI-Sim.

2.2.1 CloudSim

CloudSim [5] is a well-known simulator written in Java. The *CloudSim* simulator is an extensible framework for modelling and simulating cloud computing infrastructures and services. It allows for end-to-end cloud network architecture performance evaluation based on the topologies generated by the BRITE [27] specification. The simulator aims to provide researchers and industry-based developers with a framework to allow for system design without focusing on the low-level details of cloud-based infrastructure.

The *CloudSim* simulator consists of a multi-layered architecture made up of three core layers: *user-code*, *CloudSim*, and a *simulation engine*. The simulation engine is a low-level layer on which provisioning hosts to VMs, managing application execution, and monitoring how dynamic system states are handled. Under normal circumstances, the simulation engine is controlled by the CloudSim layer. The topmost layer, user-code, exposes the basic properties of the current cloud deployment state to the user. The layer allows for the extension of resource entities, creating different workload distributions and configurations, and trying different application provisioning algorithms. The middle layer, CloudSim, is responsible for executing the user requests on the simulation engine and gathering all analytical data from the simulation engine.

The internal communication between CloudSim entities is done using a message broker. Each entity registers within the Cloud Information Service (CIS) registry, using the `CISRegistry` class. The `CISRegistry` serves as an abstraction to communicate over the internal messaging system, using the `DatacenterBroker`. The `DatacenterBroker` class resembles a broker, acting on behalf of the user, allowing for the allocation of resources/services that meet the application *Quality of Service* (QoS) requirements.

2.2.2 GreenCloud

GreenCloud is an open-source packet-level simulator of energy-aware cloud computing data centres written in C++/OTcl [22]. The simulator views cloud computing from an energy efficiency perspective to give insights into how efficient workloads transform power into computing or data work to satisfy the demands. GreenCloud provides precise insight into the underlying server and network's energy models due to the packet-level simulation. GreenCloud is built on top of the network simulator ns-2 [31]. The simulator ns-2 provides support for the complete TCP/IP protocol stack that is implemented in all data centre components (*i.e.,* servers, switches, and links).

The energy consumption of each component is modelled as a linear function. The linear energy consumption function is altered based on which energy consumption feature each component supports. For example, in servers, Dynamic Power Management (DPM) and Dynamic Voltage and Frequency Scaling (DVFS) could be enabled to save energy at the processor core level. GreenCloud distinguishes the type of workload into three categories: Computationally Intensive Workloads (CIWs), Data-Intensive Workloads (DIWs), and Balanced Workloads (BWs). Workloads arrive following a predefined distribution (*e.g.,* Exponential or Pareto) or can be generated from network trace logs. The workload type is uniformly distributed.

2.2.3 Telco Cloud Simulator

Telco Cloud Simulator (TCS) is a simulator developed by Nokia Bell Labs that models advanced user behaviour (*e.g.,* user movement across a network) and its effect on cloud environments and distributed *Mobile Edge Computing* (MEC) subsystems [17, 16]. The development of cellular mobile communication and the increase in clients connected to wireless telecommunication networks create numerous unique challenges. Telco cloud infrastructure differs significantly from the operation of traditional IT cloud infrastructure. The requirements for telecommunications are much higher, requiring low latency, high availability, and high throughput while supporting a large number of concurrent users. In addition, traffic is much less predictable since users travel quickly within the network. TCS has been developed to deal with research related to these networks' unique characteristics and requirements.

The simulator models the dynamic and mobile user traffic in the form of *stories*. Each story represents a timeline of a user in the network, during which the user switches between different activities (*e.g.,* watching TV, calling, travelling, working), at different locations, and at different times. Each activity is assembled in a *scene* in which a user can only participate one at a time. Some scenes are location-specific; when a user moves to a scene at another location, it becomes a *moving scene*. Its travel type can further characterise each moving scene (*e.g., bike, public transport, self-driving car*), resembling different traffic patterns and requirements.

TCS is an end-to-end simulator allowing for cross-data centre cloud infrastructure simulation. Data centres can be configured with computing units, memory, and storage, interconnected with links. On top of the resources, VNF instances can be allocated and be scaled up or down accordingly. Each story can propagate through multiple data centres, and the traffic is modelled in frames instead of an event-based approach. Due to the more dynamic type of traffic, the simulator is more geared towards evaluating adaptive VNF allocation algorithms.

2.2.4 CVI-Sim

CVI-Sim is a discrete event-based simulator that originates from the research of Papagianni et al. [34], written in Java. The simulator serves as an extendable experimentation environment that allows for the evaluation of resource allocation approaches. To test real-world deployments, network topologies can be created by importing resource specification files, such as *PlanetLAB/-GENI RSpec* [37, 11]. An extended resource set provides easy access to parameters (*e.g.,* operating system, disk space, available memory) of the virtualised resources, allowing for the evaluation of the allocation algorithms.

In the paper of Papagianni et al. [34], the use of the following libraries are described: The JUNG software library was used for the underlying network graph. The JUNG library is a software library that provides a common and extendable language for the modelling, analysis, and visualisation of data represented as a graph or network [25]. The GUI of the simulator has been implemented using Swing/AWT from the *Java Foundation Classes* (JFC) framework. At last, the *CPLEX* library has been used to solve the relaxed MIP problem [18]. The Java project consists of seven packages:

- *main* — Contains the class `MainWithoutGUI` responsible for configuring and launching a headless simulation.
- *ML* — Consist of various resource allocation implementations utilising reinforcement learning. Including the parameters of the models as global constant variables.
- *model* — Contains all extendable entity classes responsible for transferring and transforming data in the substrate network and simulator.
- *monitoring* — Consists of the `Monitor` class, which can provide feedback from the simulator to certain feedback-based algorithms.
- *simenv* — Contains the parameters used by the nodes and links in the substrate network and service requests. Stored as global constant variables.
- *tests* — Examples and test cases.
- *tools* — Utility and helper classes.

The CVI-Sim simulator is structured around five important classes: `Simulation`, `Monitor`, `Substrate`, `AlgorithmNF`, and `Request` (visualised as UML in Figure 2.3). The `Substrate` class represents the substrate network consisting of all the computational and transport resources generated by the resource specification file or configured as desired. The `AlgorithmNF` class contains multiple resource allocation algorithms, which each iterate over the service requests and allocate them accordingly to the substrate network. An algorithm in the `AlgorithmNF` class can be chosen by providing a unique identifier. Finally, the `Simulation` class brings it all together and provides an abstraction on which the simulation is started using the `launchSimulation` method inside the `MainWithoutGUI` class. The `Monitor` class acts as an intermediate, providing feedback from the simulation to some of the algorithms about its effectiveness. Figure 2.4 summarises the complete simulation process.

When a simulation is launched, a new Excel file is created to output the utilisation of the substrate network. The SFC requests are then generated on a timeline following, e.g., a Poisson distribution, mimicking real-world internet traffic patterns [6]. After all the initialisation steps are completed, the actual simulation is started; iterating over each time step. During each time step, the time step's service requests are retrieved and embedded on the substrate network using the algorithm. At the end of each time step, the utilisation of the substrate network is written to the Excel file.

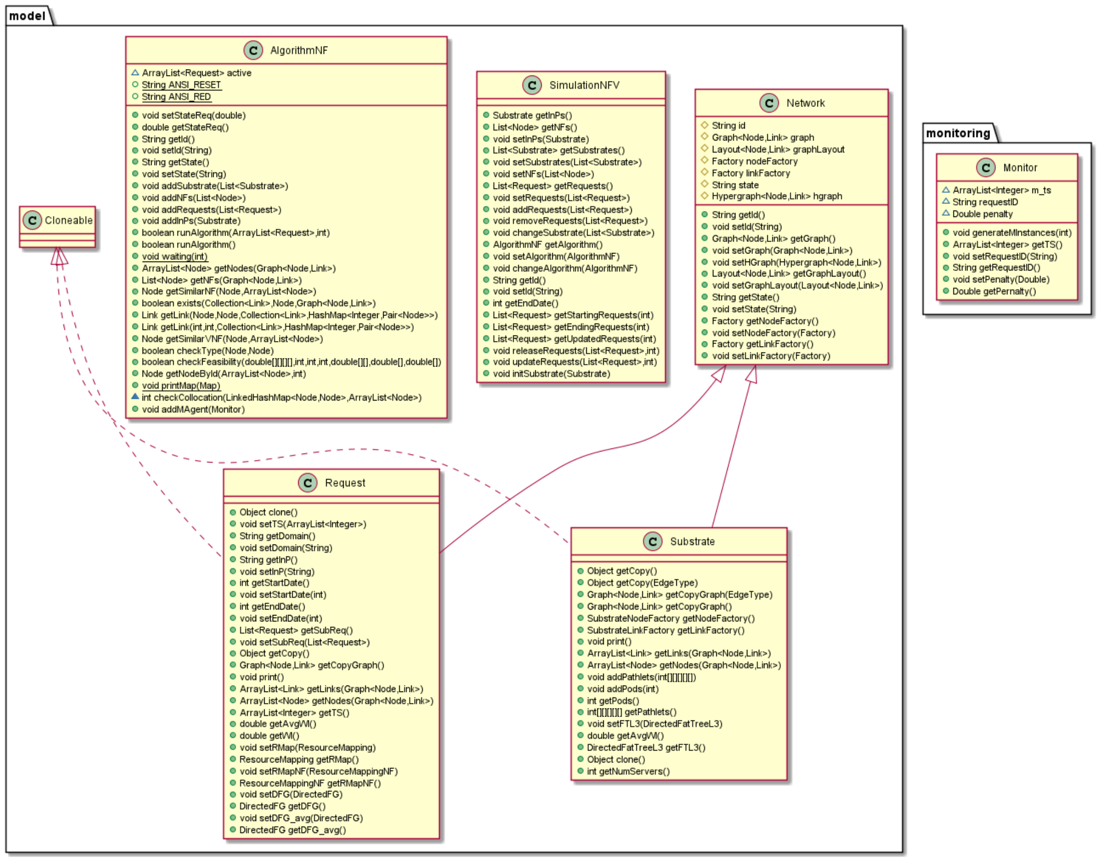

model
AlgorithmNF
ArrayList<Request> active
String ANSI_RESET
String ANSI_RED
void setStateReq(double)
double getStateReq()
String getId()
void setId(String)
String getState()
void setState(String)
void addSubstrate(List<Substrate>)
void addNFs(List<Node>)
void addRequests(List<Request>)
void addInPs(Substrate)
boolean runAlgorithm(ArrayList<Request>,int)
boolean runAlgorithm()
void waiting(int)
ArrayList<Node> getNodes(Graph<Node,Link>)
List<Node> getNFs(Graph<Node,Link>)
Node getSimilarNF(Node,ArrayList<Node>)
boolean exists(Collection<Link>,Node,Graph<Node,Link>)
Link getLink(Node,Node,Collection<Link>,HashMap<Integer,Pair<Node>>)
Link getLink(int,int,Collection<Link>,HashMap<Integer,Pair<Node>>)
Node getSimilarVNF(Node,ArrayList<Node>)
boolean checkType(Node,Node)
boolean checkFeasibility(double[][][][],int,int,int,double[][],double[],double[])
Node getNodeById(ArrayList<Node>,int)
void printMap(Map)
int checkCollocation(LinkedHashMap<Node,Node>,ArrayList<Node>)
void addMAgent(Monitor)
SimulationNFV
Substrate getInPs()
List<Node> getNFs()
void setInPs(Substrate)
List<Substrate> getSubstrates()
void setSubstrates(List<Substrate>)
void setNFs(List<Node>)
List<Request> getRequests()
void setRequests(List<Request>)
void addRequests(List<Request>)
void removeRequests(List<Request>)
void changeSubstrate(List<Substrate>)
AlgorithmNF getAlgorithm()
void setAlgorithm(AlgorithmNF)
void changeAlgorithm(AlgorithmNF)
String getId()
void setId(String)
int getEndDate()
List<Request> getStartingRequests(int)
List<Request> getEndingRequests(int)
List<Request> getUpdatedRequests(int)
void releaseRequests(List<Request>,int)
void updateRequests(List<Request>,int)
void initSubstrate(Substrate)
Network
String id
Graph<Node,Link> graph
Layout<Node,Link> graphLayout
Factory nodeFactory
Factory linkFactory
String state
Hypergraph<Node,Link> hgraph
String getId()
void setId(String)
Graph<Node,Link> getGraph()
void setGraph(Graph<Node,Link>)
void setHGraph(Hypergraph<Node,Link>)
Layout<Node,Link> getGraphLayout()
void setGraphLayout(Layout<Node,Link>)
String getState()
void setState(String)
Factory getNodeFactory()
void setNodeFactory(Factory)
Factory getLinkFactory()
void setLinkFactory(Factory)
Cloneable
Request
Object clone()
void setTS(ArrayList<Integer>)
String getDomain()
void setDomain(String)
String getInP()
void setInP(String)
int getStartDate()
void setStartDate(int)
int getEndDate()
void setEndDate(int)
List<Request> getSubReq()
void setSubReq(List<Request>)
Object getCopy()
Graph<Node,Link> getCopyGraph()
void print()
ArrayList<Link> getLinks(Graph<Node,Link>)
ArrayList<Node> getNodes(Graph<Node,Link>)
ArrayList<Integer> getTS()
double getAvgVM()
double getVM()
void setRMap(ResourceMapping)
ResourceMapping getRMap()
void setRMapNF(ResourceMappingNF)
ResourceMappingNF getRMapNF()
void setDFG(DirectedFG)
DirectedFG getDFG()
void setDFG_avg(DirectedFG)
DirectedFG getDFG_avg()
Substrate
Object getCopy()
Object getCopy(EdgeType)
Graph<Node,Link> getCopyGraph(EdgeType)
Graph<Node,Link> getCopyGraph()
SubstrateNodeFactory getNodeFactory()
SubstrateLinkFactory getLinkFactory()
void print()
ArrayList<Link> getLinks(Graph<Node,Link>)
ArrayList<Node> getNodes(Graph<Node,Link>)
void addPathlets(int[][][][][])
void addPods(int)
int getPods()
int[][][][][] getPathlets()
void setFTL3(DirectedFatTreeL3)
double getAvgVM()
DirectedFatTreeL3 getFTL3()
Object clone()
int getNumServers()
monitoring
Monitor
ArrayList<Integer> m_ts
String requestID
Double penalty
void generateMInstances(int)
ArrayList<Integer> getTS()
void setRequestID(String)
String getRequestID()
void setPenalty(Double)
Double getPernalty()

Figure 2.3: UML digram of the essential classes in CVI-Sim

2.2.5 Comparison

GreenCloud is a functionally different simulator while sharing the same goals as CVI-Sim and CloudSim, that is, testing resource allocation algorithms. The underlying packet simulator allows for far more granularity but only allows for small networks due to the computational and memory intensity of packet-level simulations. TCS does not use packet-level simulation but uses frames to represent different traffic patterns in the network, each frame specifying the size, type, and duration of the traffic. GreenCloud supports networks consisting of a few thousand nodes, while CloudSim could simulate millions of nodes [22]. TCS and CloudSim provide even more extensive networking capabilities, including multi-cloud VNF deployments spanning multiple data centres.

TCS has been developed specifically for the telecom industry, which handles more dynamic and unpredictable traffic patterns with high demand and tight constraints. Therefore, the simulator focuses more on adaptive provisioning algorithms supporting traffic mobility.

The communication between CloudSim and CVI-Sim is fundamentally different. In CVI-Sim, the algorithm directly manages the resources of the substrate network. CloudSim uses an internal message broker between the user and the manager of the substrate network. In essence, CVI-Sim is architecturally structured the same way as CloudSim but differs in the internal way of communicating. Both simulators have a low-level simulation engine layer on which the substrate network is presented. Both simulators provide entities that can be configured to represent the resources on the network and service requests. The CloudSim configuration is achieved by extension of the entities, while the CVI-Sim also allows configuration through resource specification files. The simulators differ the most in the topmost layer. The allocation algorithm is tightly integrated with the CVI-Sim simulator's simulation engine without having an abstract interface. In contrast, the CloudSim provides for each schedulable entity an interface to allow for the alternation of the allocation/provisioning policy, communicating through the internal messaging system. The simulators both provide a way to extract the utilisation parameters of the substrate network. CVI-Sim exposes many essential variables such as CPU utilisation, rejection ratio, and revenue. The CloudSim supports these variables as well and provides general insights into energy consumption. GreenCloud offers even more specific energy consumption insights by allowing low-level energy-saving features to be taken into account. The compared simulators require the allocation algorithms to be tightly integrated inside their framework, providing no external interface for the use of a remote agent.

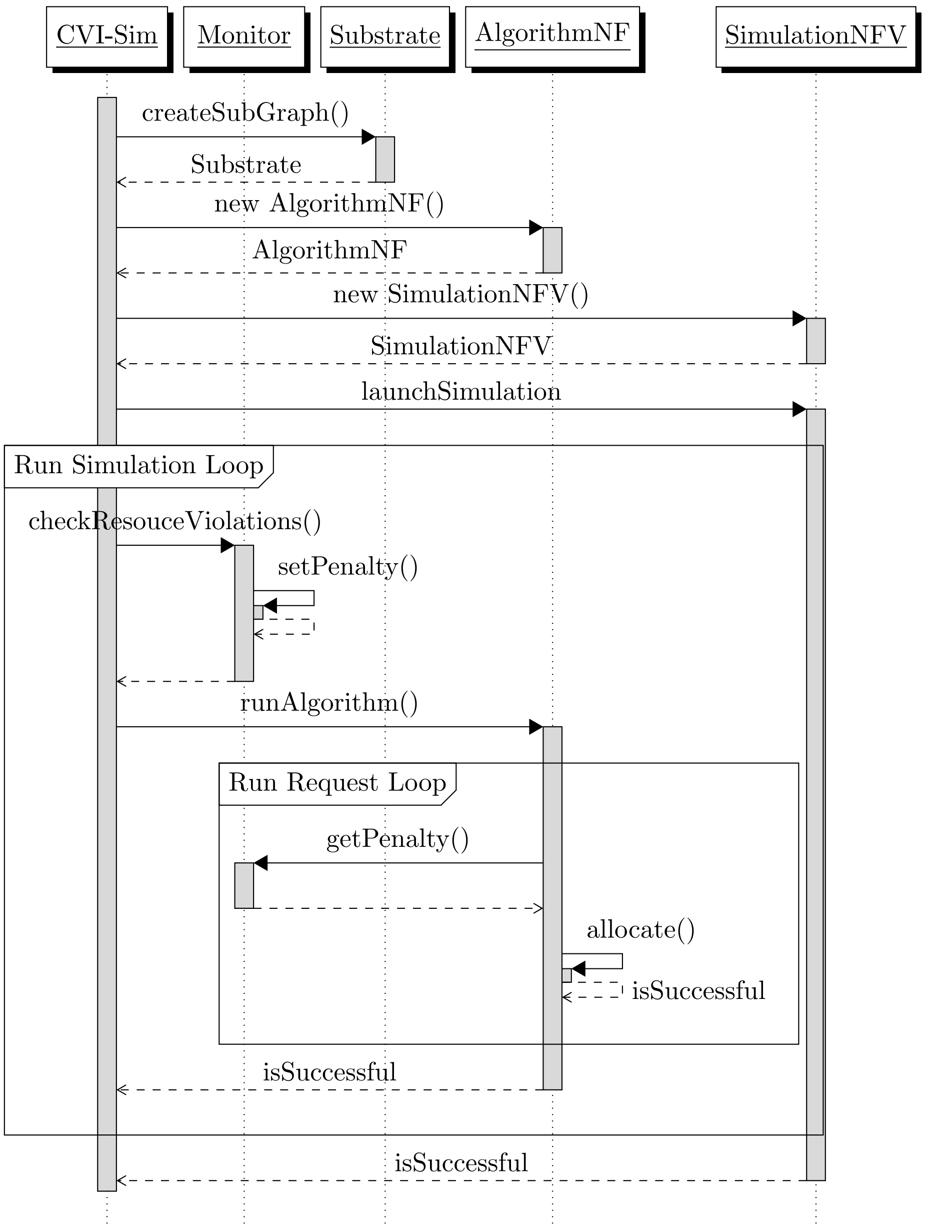


Figure 2.4: Overview of the CVI-Sim simulation process

2.3 Reinforcement learning

Reinforcement learning is an area of machine learning in which an agent learns in the form of trial-and-error, mimicking human learning behaviour [29]. The agent performs actions within an environment and receives a reward for the taken action, creating a feedback loop as shown in Figure 2.5. The agent seeks to find its optimal policy (*i.e.,* sequence of steps) by maximising its cumulative reward. The learning process can be formalised as a *Markov Decision Process* (MDP), consisting of two actors: The agent and the environment. The MDP is a discrete-time process defined as a tuple $(S, A, \mathcal{P}, \mathcal{R})$, where:

- S — is the finite set of all possible states, called the *state space.*
- A — is the finite set of actions, called the *action space.*
- $\mathcal{P}$ — is the probability $\mathcal{P}(s_{t+1} \mid s_t, a_t)$ that $a_t \in A$ during $s_t \in S$ will lead to $s_{t+1} \in S$.
- $\mathcal{R}$ — is the immediate reward received $\mathcal{R}(s_t, s_{t+1})$ after transitioning from $s_t \in S$ to $s_{t+1} \in S$.

The agent moves incrementally through the process at each time step t, by taking an action $a \in A$ during $s_t \in S$, transitioning the environment to the next state, providing the agent with the state s_{t+1} and reward r_{t+1} [41]. The MDP is often used as a framework for modelling decision making situations, to which reinforcement learning can be applied. A more extensive description of reinforcement learning and various algorithms is provided in Sutton and Barto [41]. A popular reinforcement learning algorithm is Q-learning, which has been used in recent research as a new approach to the SFC embedding problem [28, 10, 7].

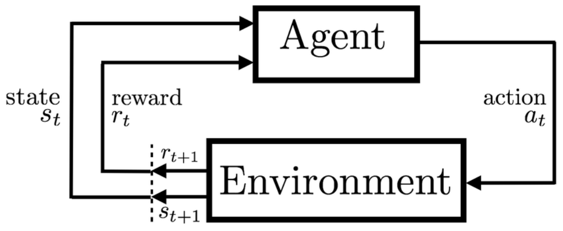


Figure 2.5: Reinforcement learning feedback loop formalised as a Markov Decision Process [41]

2.3.1 Q-learning

Q-learning is a temporal difference reinforcement learning algorithm [44, 41]. The algorithm learns by continuously evaluating and improving the *value* of an action in a given state. The algorithm is considered model-free since it does not use a model to consider possible future situations before they are experienced [41]. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances.

The value of an action a in a given state s is expressed as the *Q-value* $Q(s, a)$. The algorithm gradually learns the (near-)optimal policy by updating the Q-values until it converges to select the best action in every possible state. The Q-values are updated every episode using the Q-learning equation 2.8. The Q-learning algorithm is shown in Algorithm 1.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{future value}}}_{\text{new value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)}^{\text{temporal difference}} \tag{2.8}$$

Policy The policy maps the state to an action and is optimised to find the optimal mapping (*i.e.,* policy) [24]. The policy $\pi(a_t|s_t)$ is continuously used by the agent within the MDP. At every time step t, the agent observes an state s_t and follows the policy $\pi(a_t|s_t)$ (*i.e.,* mapping the state to an action), transitioning the agent into the next state s_{t+1} with reward r_{t+1}. The policy optimises by constantly evaluating the taken action, weighting its new value, and updating it.

There are many possible ways to select an action within a policy (*e.g.,* UCB-1, softmax, ϵ -greedy) [42], a common strategy is ϵ-greedy. The strategy selects an action greedy most of the time by taking the action with the highest Q-value at state s_t as $\text{argmax}_a\, Q(s_t, a)$. The strategy explorers by selecting an action randomly with a small probability ϵ, referred to as the exploration rate.

State-action-value function update The $Q^{new}(s_t, a_t)$ corresponds to the new state-action value function (Q-value) in state s_t after taking action a_t, receiving the reward r_t, and leading to next state s_{t+1}. The two parameters $0 \leq \alpha \leq 1$ and $0 \leq \gamma \leq 1$ are referred to as the learning rate and discount factor, respectively. The learning rate determines the importance of new information and the discount factor, indicating the importance of future rewards [28]. The policy is updated every episode using Equation 2.8, as the weighted average of the old value and the new estimated future value. Calculating the Q-value as the maximum expected future reward that the agent will receive at state s_t taking action a.

Q-table The policy of the Q-learning algorithm is often implemented on top of a lookup table, called the Q-table. The Q-table contains the Q-value of each action, a, for each possible state combination, s, indexed by $Q(s, a)$. The Q-values of the Q-table are often initialised completely random but can be initialised manually to introduce a bias.

The Q-learning algorithm is not very practical for complex problems with many states since the Q-table becomes far too large to reside in memory. To account for this issue, states are often reduced. Reduction can be done by discretisations of similar states or by representing states indirectly. In recent research, function approximation is often applied with deep learning. DeepMind has used this approach in AlphaZero to play a game of Go at a high level, defeating professional Go players [38].

Algorithm 1 Q-learning algorithm

```
1:  Initialise $Q(s, a)$, for all $s \in S, a \in A$
2:  for each episode do
3:      Initialise $s_t$
4:      for each step in episode do
5:          $a_t \leftarrow \pi(s_t)$                          ▷ Select action for current state using policy
6:          $s_{t+1} \leftarrow T(s_t, a_t)$                   ▷ Take action, transition to next state
7:          $r_t \leftarrow \mathcal{R}(s_t, a_t)$             ▷ Receive reward from environment
8:          $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot [r_t + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$   ▷ Update Q-value
9:          $s_t \leftarrow s_{t+1}$
10:     end for
11: end for
```

2.3.2 Related work

Mijumbi et al. [28] modelled a multi-agent system where each agent represents a resource (*i.e.,* node or link) in the substrate network. The node agents manage its queue, and the link agents manage the link bandwidth. The agents dynamically allocate their resources to the virtual node and links, ensuring that the resource is not underutilised. All agents have access to the utilisation of the complete substrate network and the allocated virtual nodes and links.

The algorithm uses an action space consisting of nine actions, each indicating a percentual increase or decrease in allocated resources. The state of each agent is a vector representing its allocated virtual resources. Each state is a tuple of three percentual values: Allocated resources, unused virtual resources, and unused substrate resources. The continuous percentage values have to be turned into discrete values by dividing them into chunks of 12.5%, resulting in eight different value states (3 bits). Totalling nine bits per tuple, implying a state space of $2^9 = 512$ possible states, with a total of $9 \times 512 = 4608$ state-action values.

Every time the agent has taken an action, it receives a reward. The reward value is calculated based on the monitored network state, such as link delay, packet drop, and resource utilisation. The learning algorithm then tries to converge to an optimal state, maximising single node utilisation and minimising network interruptions.

In contrast to the previous multi-agent approach, Fu et al. [10] uses a single-agent *Deep Reinforcement Learning* (DRL) approach. The single DRL agent is responsible for embedding the service function chains on the complete substrate network. In the SFC embedding processes, the allocations of the VNFs in the requested service function chain are embedded sequentially.

The learning algorithm represents the state as the currently available resources in the substrate network and the available bandwidth of the links. The action space of the DRL agent is defined as $\{1, 2, 3, \ldots, N\}$, where N represents the number of nodes in the substrate network. Each action defines to which node in the substrate network the VNF will be allocated. The reward function takes only the transmission delay of the embedded service function chain into account.

Deep Q-learning (DQL) has been used to make the algorithm more practical by allowing a large number of states. In addition, it has shown to increase stability compared to traditional Q-learning. Target networks, popularised by Mnih et al. [29], have been used to accomplish this. The target network setup uses two neural networks, one a copy of the other. One neural network, `eval_net`, is used to estimate the Q-values based on only the latest available state parameters. The second neural network, `target_net`, is used to estimate the values of the Bellman equation and is synchronised with the `eval_net` network periodically.

CHAPTER 3

Design

This chapter presents the design of the CVI-Sim extension and the remote agent interface. First, the overall architecture is explained, motivating the design decisions. Followed by the descriptions of the individual components. The design consists of three main components:

- *Messaging system* — Serves as a communication layer between the simulator and the remote agent. Providing integration in a loosely coupled and interoperable way.
- *Remote agent* — Decouples the decision-making process from the simulator, transferring the responsibility and complexity to an external entity.
- *CVI-Sim extension* — The extension of the simulator in which the decision-making process is delegated over the messaging system to the remote agent.

3.1 Architecture

In the design of the extension, the remote agent and the simulator are communicating through a messaging system, see Figure 3.1. The design decouples the decision-maker from the simulator and transfers either the admission control or the resources allocation functionality to the remote agent. Messaging systems are often used to create a modular and loosely coupled way of communication [26]. These systems allow for easy replacement of the remote agent or the simulator, providing a way to compare different simulators or agent implementations.

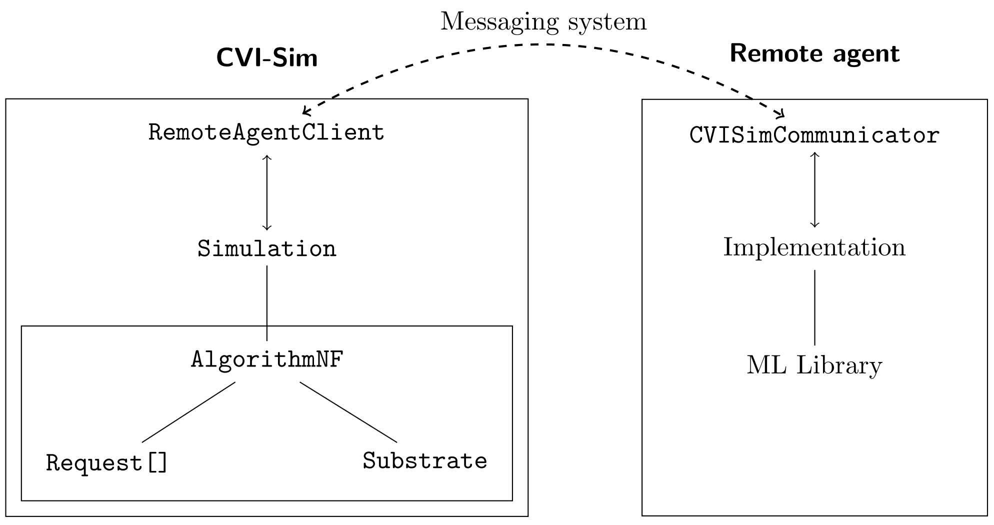


Figure 3.1: Architecture of the extension

3.2 Messaging system

Messaging systems are responsible for transferring data between applications or processes. Since the machine learning ecosystem primarily exists in Python, an integration between the Java-based simulator and the Python-based remote agent needs to be built. Messaging systems are an excellent fit for this since multiple clients often exist in different programming languages, allowing for easy and flexible integration.

Messaging systems often use the producer-consumer pattern. The producer sends messages to the system, and the consumer can subscribe to these messages in the system. A client using a messaging system is not limited to a single role but can send (produce) multiple messages and subscribe to multiple topics (consume) [23, 3]. Messaging systems are often asynchronous, which means that the client does not have to wait for a response. The asynchronous property minimises the performance impact of the new architecture's distributed nature but increases the complexity.

In the extension, the simulator and the remote agent will both function as producers and consumers. The simulator will consume the decisions of the remote agent and produce messages containing the service requests and network state. The selection of the appropriate messaging system has been made on the following requirements for the extension:

- Native Java and Python messaging support.
- Guaranteed message delivery.
- No duplicate message delivery.
- Guaranteed message ordering.
- Low complexity and ease of use.

Many messaging systems have been developed, with Apache Kafka [23], RabbitMQ [43], and ZeroMQ [2] being popular choices [19]. In the following section, the different messaging systems will be compared, and the most suitable messaging system is selected for the extension.

Apache Kafka is an open-source distributed messaging system built for high throughput, scalability, and availability developed by the Apache Software Foundation [23]. Apache Kafka offers asynchronous communication and offers a native Java and Python client. The messaging system guarantees message delivery at the cost of receiving duplicates. Kafka stores messages in partitions distributed over the nodes in the cluster. Inside the partition, the order of messages is preserved [23, 8]. Therefore, message ordering can only be guaranteed if a single node is used, which would nullify all the benefits of Kafka's distributed nature. The Kafka message broker has to be run as a separate process.

RabbitMQ is a lightweight open-source message broker built for easy deployment with advanced message routing capabilities [43]. RabbitMQ supports the AMQP [32] protocol, asynchronous communication, and a native Java and Python client. When the AMQP protocol is used, RabbitMQ guarantees message ordering in a simple setup consisting of a single queue, channel, and exchange [8]. RabbitMQ requires to be run as a separate process to connect to and communicate over.

ZeroMQ is an open-source universal asynchronous messaging library [2]. ZeroMQ is brokerless, which means no external service or daemon is needed. ZeroMQ provides no delivery or ordering guarantees other than that no partial messages can be received. ZeroMQ supports asynchronous communication and offers a native Java and Python library.

Apache Kafka has shown that it has a much higher throughput than RabbitMQ but trades in on reliability in the form of possible data loss [8]. Dobbelaere and Esmaili [8] suggest that when data loss is unacceptable, AMQP is a better choice. ZeroMQ can achieve higher throughput while maintaining lower latency than AMQP [14]. ZeroMQ has measured that 90% of measured latency values are below 1 ms under various traffic loads. With AMQP, 70% of the messages have a delay below 1 ms and about 90% below 3 ms.

Kafka and RabbitMQ have their place in distributed systems. Using one of these for the extension would significantly increase the complexity of the simulation setup. The extension does not need built-in scalability features, which can lose messages due to their distributed nature. Data loss is unacceptable for the simulation environment since it could influence the results. The simplicity of ZeroMQ is favoured since it does not need an extra component, reducing the complexity of running experiments. Although ZeroMQ does not natively provide guarantees on message delivery or message ordering, ZeroMQ would significantly reduce the complexity and minimise the added communication layer's performance impact since messages do not have to travel through an external broker. The message ordering can be implemented on the consumer side by storing out-of-order messages temporarily until the missing messages are received. Duplicate message delivery can be mitigated by storing the message offset and ignoring duplicate messages. For these reasons, it was decided to use ZeroMQ as the messaging system for the extension. Table 3.1 provides an overview of the compared messaging systems and how they meet the requirements.

Messaging system	Java client	Python client	Message delivery	Duplicates	Message ordering	Type	Complexity
Apache Kafka	Yes	Yes	Guaranteed	Yes	Partial support	Broker	High
RabbitMQ	Yes	Yes	Guaranteed	Yes	Partial support	Broker	High
ZeroMQ	Yes	Yes	No partial messages	Yes	No	Brokerless	Low

Table 3.1: Messaging systems with their characteristics related to the requirements

3.2.1 Messaging format

In CVI-Sim, the internal function calls to the algorithm will have to be replaced by messages sent over the messaging system to the remote agent. Figure 3.2 gives a visual overview of the proposed messaging structure. The following procedures in the simulator have been modelled into different messages:

- Communicating the substrate network state.
- Requesting the allocation of SFCs.
- Handling the decision of the algorithm (*e.g.,* request admission control, node mapping).
- Providing feedback to the algorithm, such as penalties or rewards.
- Initialising or resetting the state of the algorithm.

Passing the substrate network state and requesting the allocation of SFCs have been modelled into the `Request` message. The substrate network and the service request can be merged into a single entity since the allocator has to create some state representation of the network at a specific point in time. The message contains a list of `ServiceRequest` messages and a list of the `Substrate` messages (containing the network state at the time of request arrival). The list data structure provides flexibility and allows for complex allocation scenarios, such as the use of multiple substrate networks or batching service requests.

The `Substrate` message contains the substrate network as a collection of `Node` and `Link` messages, each annotated with their utilisation as a dictionary, using a list of `Parameter` messages. The `ServiceRequest` entity holds the forward-directed graph, indicating the virtualised network functions and their routing dependency with the required resource demands. The forward-directed graph is represented as a collection of `Node` and `Link` messages, reusing the message types from the `Substrate` message.

The `Request` message must be sent to the remote agent, in which the remote agent responds with a `Action` message, following a strict request-reply communication pattern. The `Action` message contains a `RequestAllocation` message for each received service request, containing three values: (i) The mandatory `success` boolean, indicating, if a service request is accepted or rejected, (ii) an optional list of `NodeAllocation` messages, structured as a simple tuple list.

Each node allocation message indicates to which substrate node a VNF in the forward-directed service graph is allocated, and (iii) an optional 2D list containing the path used by the virtual links between the VNFs in the substrate network. Each element in the list corresponds to a link in the forward-directed graph, containing the allocated path in the substrate network as an ordered list of substrate link IDs.

The `RequestAllocation` message have been designed for flexibility, allowing for different ways of using the remote agent. In a basic scenario, the node and link allocation can be ignored, and the remote agent could decide upon admitting or not using the `success` boolean, in which the actual allocation could be done using another algorithm running at the simulator. An alternative implementation could also choose to allocate nodes on the agent by including `NodeAllocation` messages and allocate the links using a greedy or multi-commodity flow allocation algorithm running at the simulator. Finally, complex scenarios can be evaluated in which the node and link allocation is an action done by the remote agent.

The feedback message has been introduced for uniformity with existing implementations inside the CVI-Sim project, relying on penalties generated by the monitoring instance. Agents should decide upon the penalty themselves using the state of the substrate network. For the purpose of evaluation in a later stage, the penalty is sent directly. At last, a `Reset` message can be sent, which is empty and serves as a signal to the agent to reset its internal state or model.

In Appendix B a detailed example is shown based on the example scenario are presented in Figure 2.2.

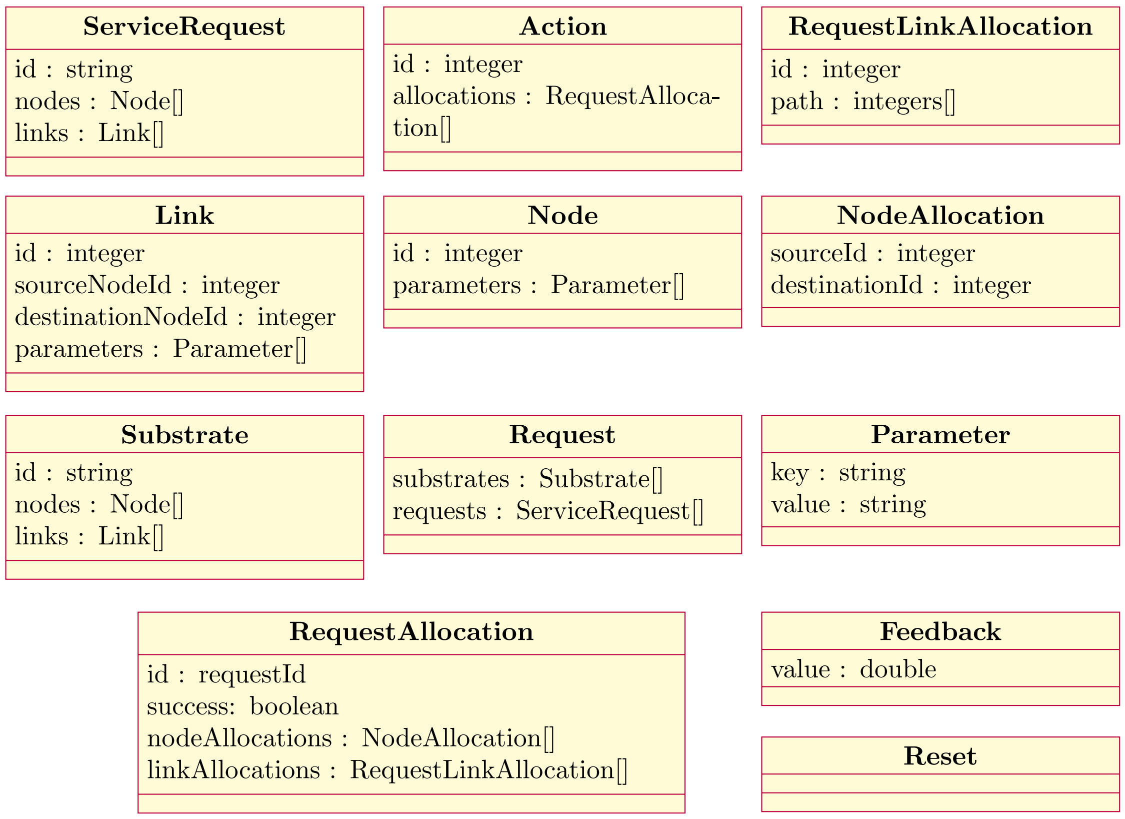


Figure 3.2: Proposed messaging format structure

Message encoding

Serialising messages can be expensive, especially in local environments where network latency is non-existent, and serialisation takes up the most significant slice of communication latency. For this reason, Google Protobuf has been chosen as the message encoding format. Google's Protocol Buffers (Protobuf) is a binary transmission format with flexible integration in multiple programming languages [13]. It has been shown that Protobuf is significantly faster than traditionally used formats (*i.e.,* XML and JSON) and produces roughly 80% smaller messages [40]. The most significant gains are seen in the serialisation and deserialisation, on which Protobuf performs serialisation almost ten times faster than XML and two times faster than JSON. In the deserialisation step, Protobuf outperforms XML by a factor of 26 and JSON by four [40].

3.3 Remote agent

The remote agent consists of two parts: The communication interface and implementation, *e.g.,* using machine learning. Using a modular communication interface, the user's implementation can use their desired programming language and libraries. Since the primary goal of the research is to give researchers using the CVI-Sim access to a vibrant machine learning ecosystem, the remote agent is designed in Python. However, the design can easily be translated to other programming languages with support for object-oriented programming.

The abstract class `CVISimCommunicator` will communicate between the user's machine learning implementation and the simulator, see Figure 3.3. The class abstracts the communication over the ZeroMQ socket and provides configuration to set up the connection. The class provides an abstract method `on_allocation_request`, which passes newly incoming `Request` messages to the user's implementation. The method requires the implementation to return an `Action`, which will be sent back to the simulator over the socket. The implementation can choose to override the protected method `on_feedback`, which is called when a `Feedback` message has been received from the simulator. At last, the `on_reset` method expects the implementation to reset its internal state.

As described in the messaging format section, the remote agent receives the substrate network states and a list of service requests in the `Request` message. The first time the `Request` message is received, the remote agent's implementation can initialise its model and return its first action to the simulator. After initialisation is done, the model can proceed to respond to incoming allocation requests.

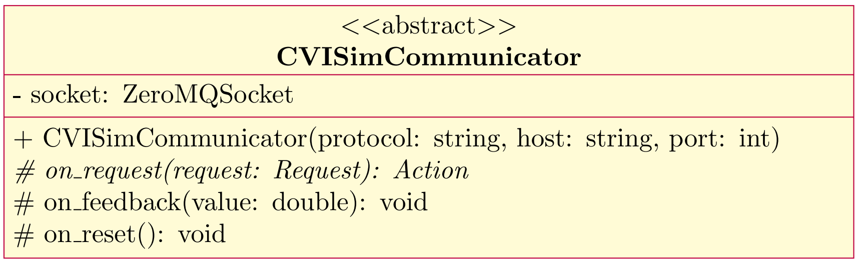
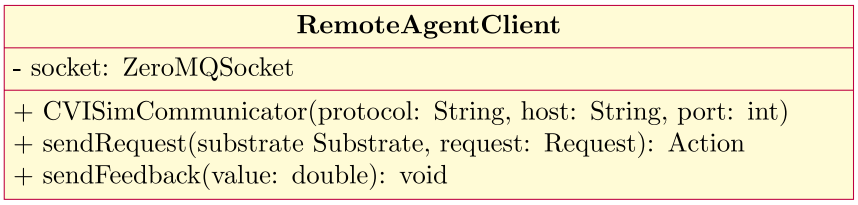

<<abstract>>
CVISimCommunicator
- socket: ZeroMQSocket
+ CVISimCommunicator(protocol: string, host: string, port: int)
# on_request(request: Request): Action
# on_feedback(value: double): void
# on_reset(): void

Figure 3.3: UML representation of the abstract `CVISimCommunicator` class

3.4 Simulator

CVI-Sim will have to be extended with an interface to support the remote agent. Using the interface, the remote agent must be able to execute its actions on the simulator. The simulator must be able to use the interface to communicate its substrate network state, service requests, and feedback to the remote agent. The interface will be built as a messaging client, which serves as an abstraction to communicate over the messaging system, described in the next paragraph.

Therefore, as described in the previous section, a new class `RemoteAgentClient` will be created in the simulator to abstract the communication over the ZeroMQ socket to the remote agent, see Figure 3.4. The class will function as an intermediate proxy. The class will not require the implementation to override specific methods but will function as a client to the remote agent. The client allows for flexible integration in existing algorithms in the `AlgorithmNF` class. The `RemoteAgentClient` asks the remote agent for an action by sending an allocation request (including its network state) and waits for the action to come back. Based on the received action, the simulator can alter its network state and report the impact of the allocation. The process is repeated until the simulation has finished. The simulator can provide feedback to the remote agent using the `sendFeedback` method. At last, the `sendReset` method can be called to reset the internal state of the remote agent. Feedback and resets can be communicated outside of the request allocation flow.



RemoteAgentClient

- socket: ZeroMQSocket

+ CVISimCommunicator(protocol: String, host: String, port: int)
+ sendRequest(substrate Substrate, request: Request): Action
+ sendFeedback(value: double): void

Figure 3.4: UML representation of the `RemoteAgentClient` class

CHAPTER 4

Implementation

This chapter describes the implementation of the design presented in the previous chapter. First, the usage of the messaging system in both components is described, followed by the description of the CVI-Sim extension and implementation of the remote agent communication interface.

4.1 Messaging system

The ZeroMQ messaging system operates over a local TCP connection. The communication pattern features a stricter implementation of the publisher-subscriber pattern. A request-reply pattern is used between the simulator and the remote agent to ensure the correct ordering of the messages. The remote agent takes the role of the replier, thereby responding only to incoming requests made by the requester, the simulator. In addition, the simulator can also send feedback and request reset at any time outside of the request-reply pattern, separate from the allocation requests. The remote agent does not respond to the feedback or reset messages.

ZeroMQ provides a way to enforce messaging patterns by specifying a socket type. A request-reply pattern could be accomplished by defining the CVI-Sim socket as the *requester* (`REQ`) and the remote agent as the *replier* (`REP`). This pattern enforces the remote agent to only respond to messages received from the simulator. It limits the simulator only to send a message after receiving a response (excluding the initial message). The messaging pattern would be ideal as it increases error detection and improves reliability. However, since multiple message flows (*i.e.,* request allocation and feedback/reset) are used and could be sent concurrently, the request-reply socket type would break and therefore can not be used. As an alternative, the `PAIR` socket type is used. The `PAIR` socket type enforces an exclusive pair pattern, which restricts the communication to two peers and allows bidirectional communication without restrictions on ordering.

Google's Protocol Buffers (Protobuf) has been used as the messaging format between the two communicating parties, allowing for an efficient and flexible messaging format across various programming languages. As proposed in Figure 3.2, the messages have been modelled using the protobuf messaging format language (see Appendix A). Using the Protocol Buffers compiler, `protoc`, the protobuf messages are compiled into their native messaging models in Java and Python used by the implementations, respectively.

The messaging system uses two distinct messaging flows; the request allocation flow and the feedback/reset flow, each with a different message format. ZeroMQ and Protocol Buffers do not natively support message type distinction. Determining the message type is crucial to deliver the message correctly and avoid deserialisation errors. Naively deserialising messages could be tried, and in case of an error, the other message format could be tried, wasting processing power. A more sophisticated and efficient solution is implemented to avoid this, denoted as *message envelopes.* With a message envelope, each message is prefixed with a fixed-length message header indicating the message data type format as shown in Figure 4.1. On the arrival of a new message, the receiver first reads the fixed-length message header, determines the message format, parses the message body, and delivers the message to the correct flow.

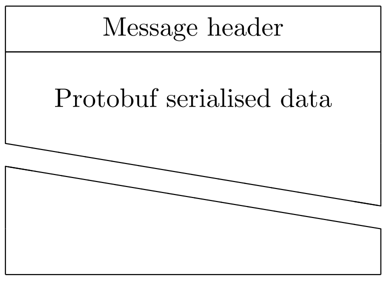


Figure 4.1: Message envelop example

4.2 Simulator extension

The CVI-Sim extension uses the `JeroMQ` library[1], which is a pure Java implementation of the low-level `libzmq` library[2]. The `RemoteAgentCommunicator` class, as presented in the design, abstracts away the communication over the ZeroMQ socket. When the `RemoteAgentCommunicator` class is constructed a new ZeroMQ socket configured with the `PAIR` flag. The socket can be configured as desired by providing the protocol, host and port of the socket. The socket is then opened, allowing the remote agent to connect to the simulation environment.

Requesting the allocation of service requests to the remote agent can be done by sending the current substrate network state and the service requests using the `sendRequest` method. The method uses a two-step process to request an allocation from the remote agent to mimic the request-reply pattern:

- **Phase 1: Sending the message**

  The method parameters `Substrate` and `ServiceRequest` are first translated into their respective generated protobuf Java models, using the static `createRequest` method from the `MessagesUtils` class. The `MessagesUtils` class is a collection of utilities for translating common CVI-Sim models to their respective protobuf generated classes. The message header is then first sent over the socket, indicated by the `SNDMORE` flag, to signal the receiver that more message parts are coming. Finally, the protobuf message model is converted to a byte array and sent over the socket, completing the sending phase.

- **Phase 2: Receiving the response**

  The sender now switches its role and listens to the socket for incoming messages. Listening is done in a blocking way to ensure correct ordering. Once a message is received as a byte array, it is serialised into a protobuf `Action` message. At last, the received `Action` message is returned to the function caller, ending the two-phase process.

Sending feedback to the remote agent can be done by using the `sendFeedback` method. Sending feedback is a single step process since no response from the remote agent is expected. First, the message header is communicated, indicated with `SNDMORE` flag. At last, the feedback value is converted to a byte array and send over the socket. The reset message can be sent using the `sendReset` method, which only communicates the message header with an empty message body.

4.3 Remote agent

The implementation of the `CVISimCommunicator` class serves as the communication interface between the simulator and the user's implementation, removing additional complexity from the user's implementation.

[1]`https://github.com/zeromq/jeromq`
[2]`https://github.com/zeromq/libzmq`

The `CVISimCommunicator` class uses the `PyZMQ`[3] library, providing Python bindings to the low-level `libzmq` library[2]. The ZeroMQ socket is initialised upon the construction of the class and can be configured by providing optional parameters in the constructor (configuring the protocol, host, and port). To start communicating with the simulator and receive messages, the user needs to call the `start` method.

Inside the `start` method, messages are read from the internal ZeroMQ socket. Whenever a new message is received, the message header is read first, followed by reading the message body. Based on the message header, the message body is serialised into the correct message type. Three types of messages can be received from the simulator:

- **`Request`** — The received request message is passed to the abstract `on_allocation_request` method. The implementation of the `on_allocation_request` method is required to return a `Action` message, which is sent back over the ZeroMQ socket to the simulator. Enforcing the request-reply pattern required for the request allocation message flow.
- **`Feedback`** — When a feedback message is received the contained value is directly passed to the `on_feedback` method. The implementation is not required to respond to a message, therefore ignoring any return value.
- **`Reset`** — On the arrival of a reset message, no deserialisation is performed since the message body is empty. After receiving a `reset` message, the `on_reset` method is called, expecting the implementer to reset its internal state. The implementation is also not required to respond to a message, therefore ignoring any return value.

The `CVISimCommunicator` does not provide a message envelope when sending back a message to the simulator. Using a message envelope for messages sent from the remote agent to the simulator is not required since only a single message type (the `Action` message) is sent back to the simulator. Including a message header would increase bandwidth, complexity and slightly degrade the performance.

[3]`https://github.com/zeromq/pyzmq`

CHAPTER 5

Test Implementation

The test implementation has been developed for the evaluation of the extension, described in the next chapter. This chapter describes the test implementation using the newly introduced components from the previous chapters.

The original CVI-Sim project contains various resource allocation approaches implemented in Java, such as greedy allocation, numerical optimisation, and machine learning. These implementations in the CVI-Sim project have been used in research for benchmarking [36, 7]. The test implementation is based on the reinforcement learning approach used in the research of Papagianni et al. [36] and De Vleeschauwer et al. [7], which has been implemented in the `QLearnBelief` class.

The implementation consists of two components, as presented in the design. In the simulator, the newly developed `RemoteAgentClient` interface is used to communicate to the remote agent. The simulator uses the same greedy allocation strategy as the reference implementation. The remote agent consists of a Q-learning algorithm, implemented on top of the `CVISimCommunicator` class.

5.1 Simulator

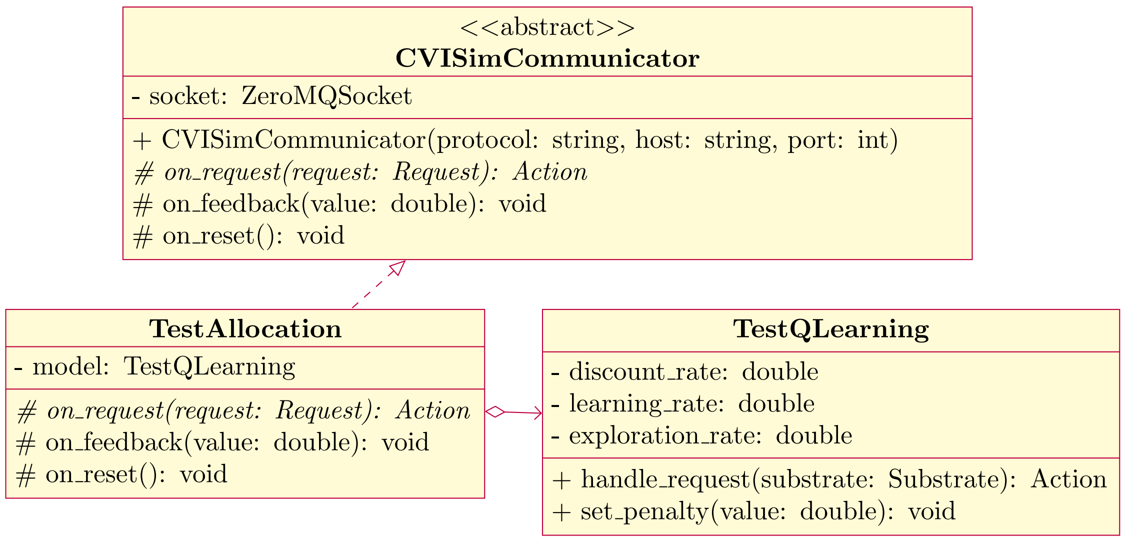
The newly created `RemoteAgent` class is responsible for the communication to the remote agent and the greedy allocation. The class exposes similar methods as the native reinforcement learning algorithms for easy integration in the `AlgorithmNF` class. The `AlgorithmNF` class is a collection of different allocation strategies and executes the specified algorithm on the allocation requests by exposing the `runAlgorithm` method. The `runAlgorithm` method is called by the simulator every simulation time step. On construction of the `AlgorithmNF` class an algorithm identifier can be provided, *e.g.,* 'RL' for using the Java-based `QLearn` implementation. The test implementation uses the 'RA' identifier, short for *remote agent*.

The `RemoteAgent` class uses the `RemoteAgentClient` internally and handles the greedy node allocation using the `greedy` method from the `QLearnBelief` class. The class exposes similar methods as the `QLearnBelief` class. The `placeloads` method performs the actual node allocation by first asking the remote agent if the request can be embedded and providing a node mapping based on the remote agent's response, using greedy allocation. The allocation request and a list of substrate networks have been added as parameters to the method, required to be sent over to the remote agent. The `setCurrentPenalty` method exposes a way for the simulator to provide feedback to the algorithm. The implementation of the method communicates the penalty to the remote agent, using the internal `RemoteAgentClient`.

The `RemoteAgent` instance is initialised as a private class variable in the `AlgorithmNF` class when the identifier is specified on the construction of the class. The private `NFplacement_RL` method inside the `AlgorithmNF` class is responsible for executing the correct reinforcement learning algorithm on the available allocation requests. By including the remote agent instance into the method, the same logging output format can be reused, making the comparison in the evaluation stage easier.

5.2 Remote agent

The remote agent's side of the test implementation contains the complex part of the setup. Implementing the Q-Learning algorithm, which originally resided directly inside the simulator. Two new classes have been constructed for the test implementation. First, the `TestAllocation` class, extending the abstract `CVISimCommunicator` class. Secondly, the `TestQLearning` class, containing the Q-learning algorithm. Figure 5.1 shows an overview of the remote agent test implementation in UML. The `TestAllocation` class implements the methods of its superclass, passing on messages to the algorithm and returning the results. The initialisation of the Q-learning state is delayed until the first allocation request is received so that the algorithm can use the substrate network to build its state with the correct dimensions.



<<abstract>>
CVISimCommunicator
- socket: ZeroMQSocket
+ CVISimCommunicator(protocol: string, host: string, port: int)
# on_request(request: Request): Action
# on_feedback(value: double): void
# on_reset(): void
TestAllocation
- model: TestQLearning
# on_request(request: Request): Action
# on_feedback(value: double): void
# on_reset(): void
TestQLearning
- discount_rate: double
- learning_rate: double
- exploration_rate: double
+ handle_request(substrate: Substrate): Action
+ set_penalty(value: double): void

Figure 5.1: UML diagram of remote agent implementation

5.2.1 Learning Algorithm

The `TestQLearning` class houses the Q-learning algorithm, based on the `QLearnBelief` reference implementation. The algorithm performs request admission control based on the state of the substrate network. Newly incoming requests can be passed on to the model by using the public method `handle_request`, returning the decision of the model. The implementation of the algorithm is done as follows and shown in Algorithm 2.

Q-learning The algorithm uses three parameters: discount rate γ, learning rate α, and exploration rate σ $(0 \leq \gamma, \alpha, \sigma \leq 1)$. The parameters can optionally be adjusted on the construction of the class.

Action space The action space consists of two actions, accept and reject, represented as integers respectively . The Q-table is a $N \times N \times 2$ table, where N is the number of nodes in the substrate network, and the number 2 corresponding to the size of the action space.

State space The state space is composed of two discrete components based on the utilisation of the substrate network: (i) s_1 being the number of servers that have a CPU utilisation above a preset threshold θ_l, and (ii) s_2 being the number of servers with an average residual lifetime above a preset threshold θ_d. The state components are constrained to $0 \leq s_1, s_2 \leq N$, therefore used as an index in the Q-table. The representation of the state space is used to introduce a bias. The Q-values of the Q-table are linearly initialised, serving as a bias to encourage request acceptance.

Replay learning The model uses experienced replay learning to mitigate correlation between decisions and to learn from rare states. The technique stores the agent's experience at every time step in the U-table called the *replay memory.* The U-table has the same dimensions as the Q-table but with all values initialised at zero. The replay memory is updated every request allocation using the temporal difference component of the Q-learning Equation 5.1, visible on line 7 in Algorithm 2. The values of the replay memory are transferred to the Q-table after an episode has passed. Each episode contains 500 requests. Due to the multidimensional nature of the data structure, tensors are used to perform the calculations efficiently using NumPy [15].

Reward The reward in the Q-learning Equation 5.1 diverges slightly from the traditional Q-learning Equation 2.8. The reward r is composed of the sum of two components: The model's immediate reward r_m and the penalty p. The value of r_m is determined directly by the model, defined as 1 if a request is allocated and 0 otherwise. The second component, p, is provided as feedback from the simulator after every allocation request.

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \left((r_m + p) + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right) \tag{5.1}$$

Policy Upon the arrival of a request in the `handle_request` method, the state is determined and used as a multidimensional index in the Q-table. Resulting in an array containing the Q-values for each action, given that state. The optimal action is then retrieved by selecting the index with the highest Q-value, using NumPy's `argmax` function to allow for vectorisation of the operation. To allow for exploration, the model can pick a random action with a probability of σ, or otherwise proceed with the selected optimal action. The action selection procedure is visible on line 14 in Algorithm 2.

Algorithm 2 Q-learning algorithm of the remote agent

```
 1: Q ← initialised linearly
 2: U ← initialised empty
 3: for each request do                                   ▷ Received requests from the simulator
 4:     $p$ ← collect penalty
 5:     $a$ ← GETACTION(state)
 6:     $r$ ← if $a$ is accepted then 1 else 0              ▷ Reward if request is embedded
 7:     $U[state][a] \leftarrow U[state][a] + (r+p) + \gamma \cdot \max Q[state] - Q[state][a]$
 8:     if episode has passed then
 9:         $Q \leftarrow Q + \alpha \cdot U$
10:         U ← reinitialised empty
11:     end if
12:     return $a$
13: end for
14: function GETACTION(state)
15:     $a \xleftarrow{R} \{0, 1\}$
16:     if $\mathcal{U}(0,1) \geq \sigma$ then                ▷ Explore new action, otherwise find best action
17:         $a \leftarrow \operatorname{argmax}_x Q[state][x]$
18:     end if
19:     return $a$
20: end function
```

CHAPTER 6

Performance Evaluation

This chapter describes how the extension of the CVI-Sim is evaluated using the test implementation and a reference implementation. First, the testing and evaluation methodology is described, outlining the steps taken to validate the correctness of the extension, supplemented by a hypothesis. At last, the requirements, criteria, and error margins for a correctly working implementation are stated.

6.1 Evaluation environment

The test implementation described in the previous Chapter (5) has been created to validate the extension's correctness and measure the extension's performance impact. The test implementation is compared to the original Java-based `QLearnBelief` implementation, serving as the *reference implementation.* The `QLearnBelief` implementation was created and used in the research of Papagianni et al. [36] and De Vleeschauwer et al. [7]. The experiments use the same experimental setup mentioned in the papers to reproduce their findings and benchmark the test implementation.

The research of De Vleeschauwer et al. [7] evaluated two types of scenarios. The first scenario measures the efficiency and stability of the embedding approach. The second scenario measures the plasticity and convergence of the approach, testing its ability to adapt to changing conditions. The scenario introduces an extra obstacle by increasing the amount of inbound traffic by approximately 30% in the last half of the simulation (5000 requests).

The experiments have been configured as follows, similar to De Vleeschauwer et al. [7]. The parameters of the Q-learning model were initialised with $\alpha = 0.005$, $\gamma = 0.7$, and $\sigma = 1\%$ and the state thresholds as $\theta_l = 10\%$ and $\theta_d = 10,000$. At the start of each experiment, a substrate network was generated as a 3-layer fat-tree topology. In the network, 36 nodes were available, of which 16 servers are available for allocation. Each server has access to 8 cores running at 2 GHz. The initial utilisation of the servers is uniformly distributed $\mathcal{U}(0.4, 0.7)$. Each experiment consists of $10,000$ service requests generated according to a Poisson time distribution. Subsequently, for each request, a forward direct graph is generated, based on three templates: (i) A service chain that handles traffic routed through a particular sequence of VNFs, (ii) a service chain in which a particular VNF splits the traffic, *e.g.,* for the purpose of load balancing, and (iii) a service chain that handles traffic differently on a per path basis. The number of requested VNFs in the service requests is uniformly distributed $\mathcal{U}(5, 10)$. Their inbound traffic rates are uniformly distributed $\mathcal{U}(10, 100)$ Mbps and used to derive the computational demand for each VNF, modelled as the number of cycles per packet. A total of ten experiments per scenario were performed.

For every experiment, to correctly compare the results of the two implementations within a single experiment, the same substrate network and service requests are used. During the experiment, at the end of each simulation time step, aggregate results are collected to provide insights into the substrate network state and how service requests are handled.

6.1.1 Testing platform

The experiments were run on a Windows 10 Pro (20H2) system with an Intel i7 10700K processor running at 5 GHz, using 32 GB of RAM. Python version 3.8.9, and NumPy [15] version 1.19.5 are used in the test implementation. The CVI-Sim extension was compiled using the Java 8 SDK.

6.1.2 Hypothesis

It is expected that the test implementation will perform similar to the reference implementation. The test implementation employs the same Q-learning model, with identical parameters, as used in the reference implementation. The implementation differs in the programming language used, while communication is established over a socket with the simulator. At the start of the simulation, it is expected that the results of the implementations differ slightly since the models use randomness to find the optimal policy. After a certain amount of requests have passed, the implementations should converge to the same admission control policy. The test implementation is expected to have a slower request allocation run time since communication over the socket adds extra latency. Additionally, Python is an interpreted programming language imposing a slower code execution time than a compiled programming language such as Java. However, this slowdown happens at a very small scale and should not impact the run time.

6.2 Performance metrics

The results of the experiments are analysed to show a similarity between the two implementations. Each experiment generates multiple time series data sets of evaluation metrics. The following subset of the collected metrics are used for the evaluation of the implementation, each parameter aggregated in episodes of 500 requests:

- **Rejection ratio** is the percentage of service requests that have been rejected.
- **Violation ratio** is the ratio of resource violations and service requests. A resource violation occurs when the capacity of a resource in the substrate network is exceeded.
- **Server utilisation** is the average CPU utilisation of the substrate servers.
- **Non-collocation ratio** is the percentage of SFCs whose VNFs are embedded on multiple servers.
- **Request allocation run time** is the time taken by the algorithm to allocate a request.

The similarity will be measured using the absolute difference between the measurement parameters of the implementations. The metric can be used since it is expected that both implementations with the same state and parameters produce similar output per simulation time step.

6.3 Results

This section presents the results of the experiments as described in Section 6.1. Initially, the rejection and violation ratios of both implementations are presented and compared, highlighting their similarity as well as their differences. Secondly, the CPU server utilisation and the SFC non-collocation ratio of the implementations are compared. At last, the measured request allocation run time of both implementations is shown.

6.3.1 Rejection and violation ratio

The measured rejection and violation ratio of the experiments is shown in Figure 6.1. The implementations are compared per ratio in Figure 6.2 using the absolute difference per episode.

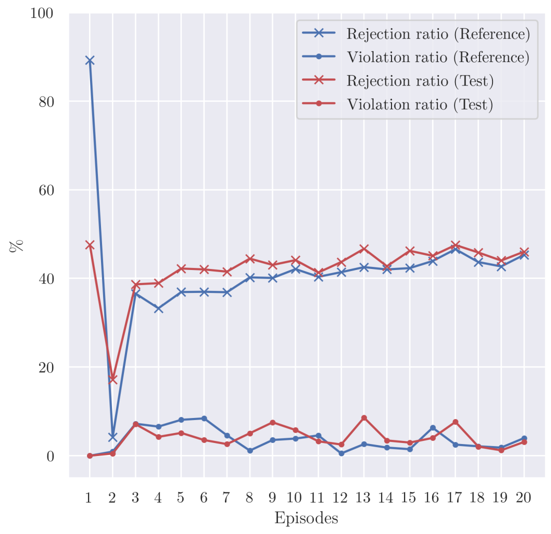
The implementations show a similar trend in the rejection and violation ratio due to the small observed difference in Figure 6.2. In the efficiency scenario, both implementations converge to an average rejection ratio of roughly 40% and an average violation ratio of approximately 5% after 20 episodes have passed (see Figure 6.1a). During the plasticity scenario, a similar pattern is observed in Figure 6.1b. Before the traffic is increased, the implementations converge to a rejection ratio of roughly 40% and afterwards converge to around 60%. The violation ratio exhibits the same pattern, converging to around 5% in the first phase and afterwards climbing to a violation ratio slightly below 20%.

Rejection ratio

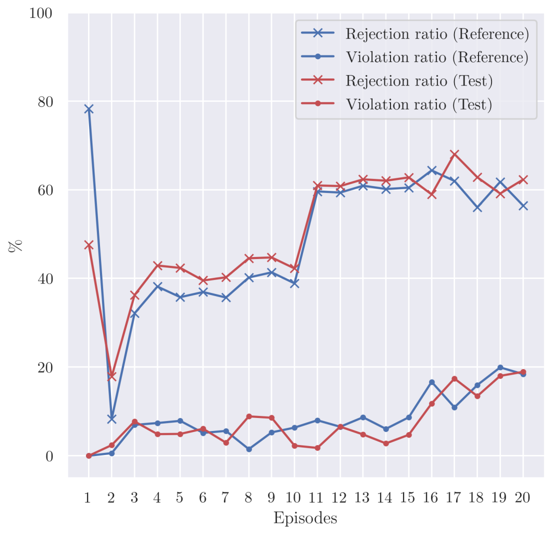
The implementations converge quickly to a rejection ratio of approximately 40% in both scenarios after three episodes have passed, visible in Figure 6.1. A high rejection ratio is observed during the first episode, with a significant difference of roughly 25% between the implementations. The implementations recover quickly in the second episode in a similar trend by accepting more requests, resulting in violations. In response, the implementations start to reject more requests in episode 3, beginning to converge towards the rejection ratio of 40%. The plasticity scenario shows the same trend as the efficiency scenario until episode 10 in Figure 6.1b. In episode 10, the rejection ratio of the implementations jumps to approximately 60% caused by the dynamic traffic increase. Afterwards, the implementations fluctuate around 60%. In contrast, during the efficiency scenario, the rejection ratio of the implementations tends to stabilise and approach each other more over time.

Violation ratio

The implementations show a similar trend in the violation ratio during both scenarios in Figure 6.1. The first resource violations are taking place in episode 2 in conjunction with the steep decrease in rejections. During the efficiency scenario, the violation ratios then start to oscillate slightly around each other between episodes 3 and 11 in Figure 6.1a. After episode 11 in the efficiency scenario, the violation ratios stabilise and ultimately become almost identical after episode 18. In contrast, during the plasticity scenario in Figure 6.1b a similar pattern is observed until episode 10, after which the violation ratio destabilises and increases until the last episode.

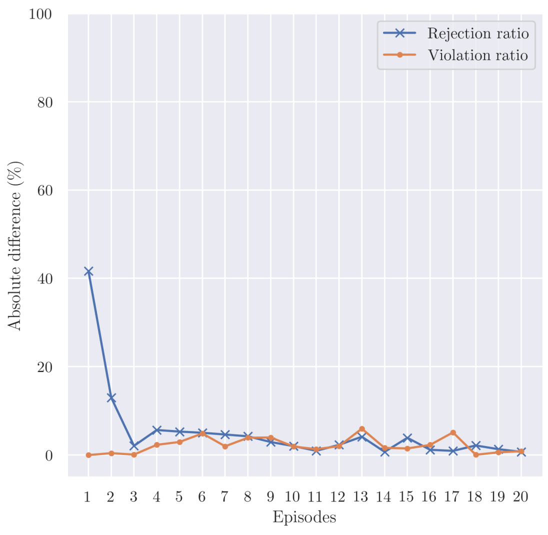


(a) Efficiency and stability scenario

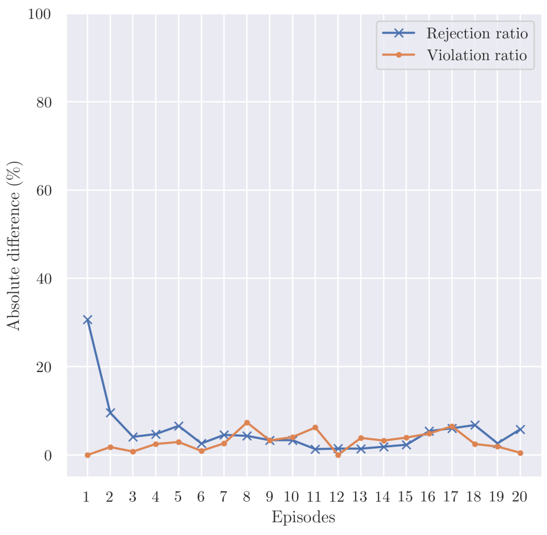


(b) Plasticity and convergence scenario

Figure 6.1: Average rejection and violation ratio per episode



(a) Efficiency and stability scenario



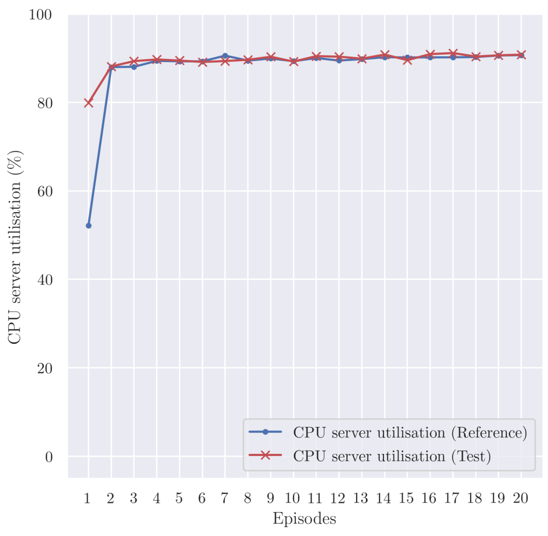
(b) Plasticity and convergence scenario

Figure 6.2: Absolute difference in rejection and violation ratio per episode

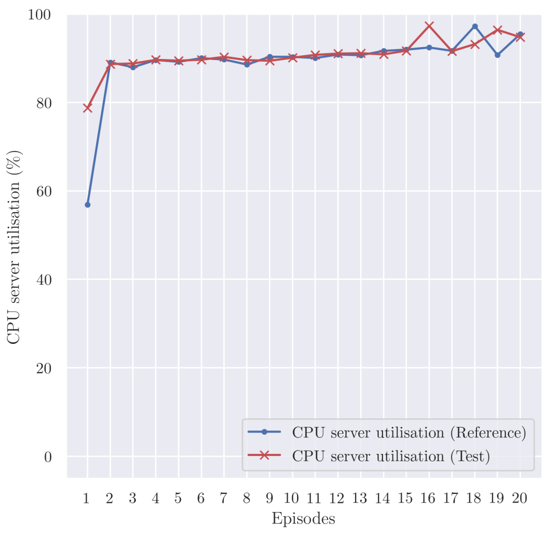
6.3.2 CPU server utilisation

The average CPU server utilisation per episode of each scenario is shown in Figure 6.3. Figure 6.4 provides a general overview of the CPU server utilisation during the experiments.

The average CPU server utilisation of both implementations has an identical overall trend in both scenarios. The implementations converge to an average CPU server utilisation of roughly 90%. A significant difference is seen in the first episode. The utilisation of the test implementation is approximately 25% higher, corresponding to the difference in the rejection ratio shown in Figure 6.1. The average CPU server utilisation during the plasticity scenario in Figure 6.3b fluctuates slightly during the end of the second phase after the dynamic traffic increase.

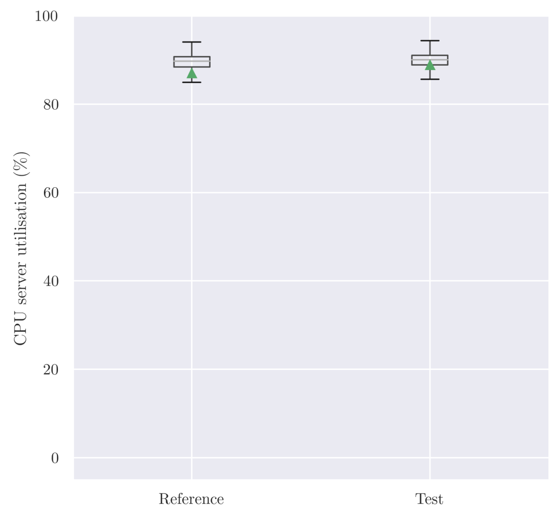


(a) Efficiency and stability scenario

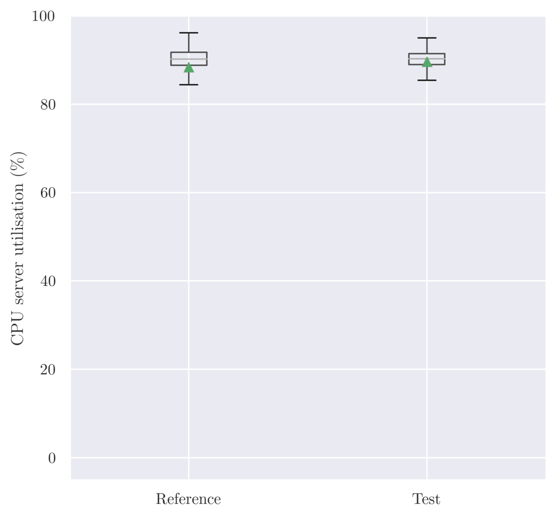


(b) Plasticity and convergence scenario

Figure 6.3: Average CPU server utilisation per episode



(a) Efficiency and stability scenario



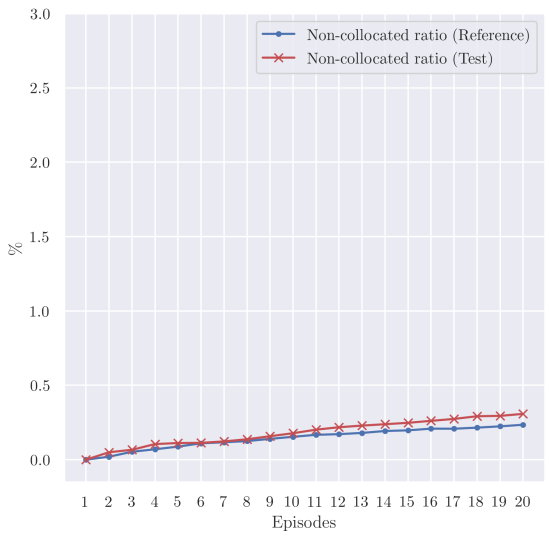
(b) Plasticity and convergence scenario

Figure 6.4: Overview of the CPU server utilisation across all experiments

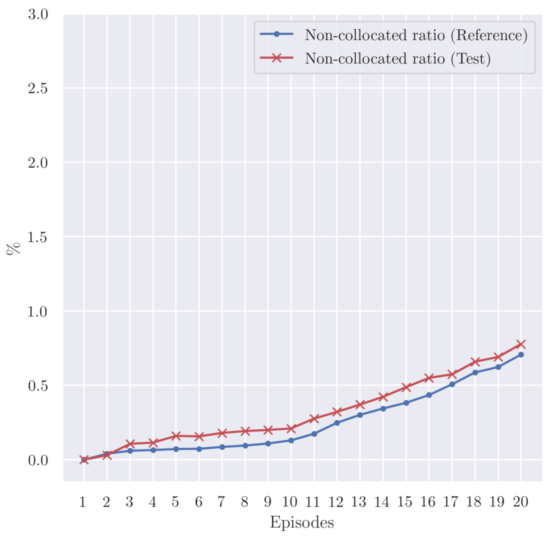
6.3.3 SFC non-collocation ratio

The average SFC non-collocation ratio per episode of the scenarios is shown in Figure 6.5. Figure 6.6 provides a general overview of the SFC non-collocation ratio measured across all experiments.

The SFC non-collocation ratios of both implementations demonstrate almost identical behaviour during the scenarios. The SFC non-collocation ratio increases due to the fragmentation of resources over time, by admitting expiring requests. During the efficiency scenario in Figure 6.5a, the implementations converge to a non-collocation ratio of approximately 0.25% and approximately 0.75% in the plasticity scenario (see Figure 6.5b). The test implementation shows a slight upward deviation towards the end of the efficiency scenario compared to the reference implementation in Figure 6.5a, but at a tiny subpercentage scale. Similarly happening in an overall trend during the efficiency scenario in Figure 6.5b. The behaviour during the plasticity scenario is almost identical to the efficiency scenario until the dynamic traffic increase in episode 10. After the increase, the average SFC non-collocation ratio starts to climb faster, due to the increase in computation requirements derived from the inbound traffic.

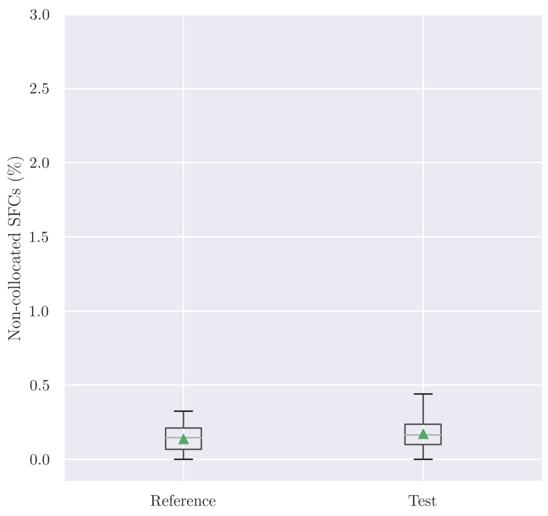


(a) Efficiency and stability scenario

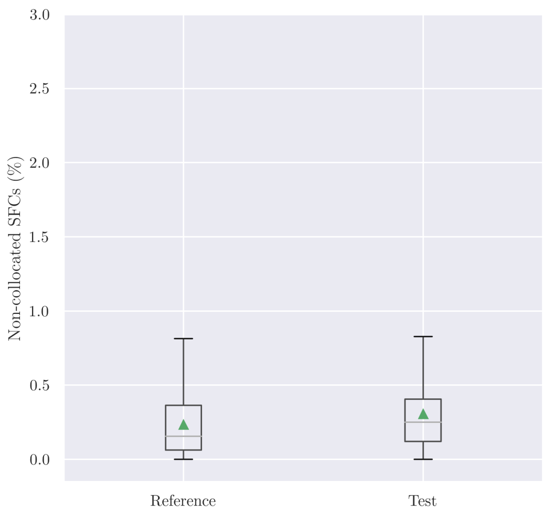


(b) Plasticity and convergence scenario

Figure 6.5: Average SFC non-collocation ratio per episode



(a) Efficiency and stability scenario



(b) Plasticity and convergence scenario

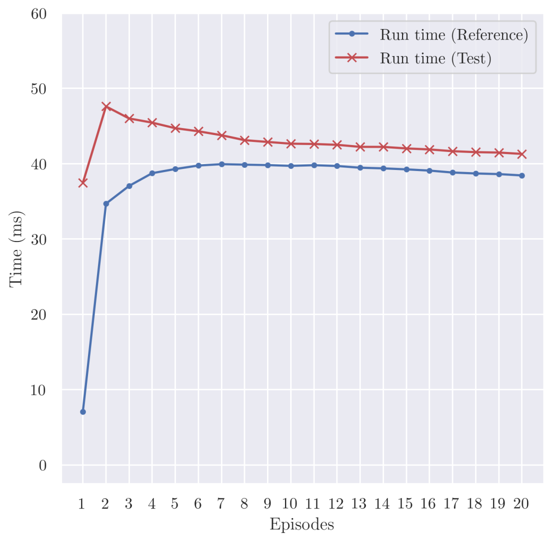
Figure 6.6: Overview of the SFC non-collocation ratio across all experiments

6.3.4 Request allocation run time

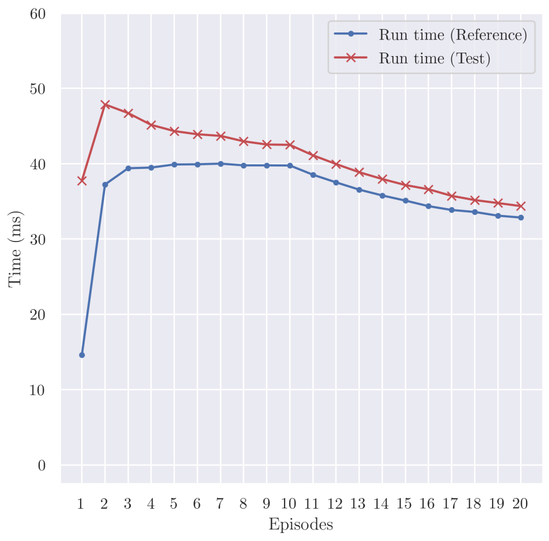
The average request allocation run time per episode of each scenario is shown in Figure 6.7. Figure 6.8 provides a general overview of the request allocation run time measured across all experiments.

The run time of the implementations shows a similar trend in both scenarios, with the test implementation being shifted slightly upwards. The average run time of the test implementation converges to an increase of roughly 3 ms during the efficiency scenario in Figure 6.7a, resulting in an average slowdown of 7% per request, mainly attributed to the introduced communication layer overhead. The plasticity scenario shows the same trend in Figure 6.7b until the dynamic traffic increases at episode 10, resulting in a decrease in run time. During the decrease, the test implementation approaches the run time of the reference implementation closer over time, converging to approximately 35 ms in the final episode. The overall distribution in Figure 6.8

shows that the decrease imposes no slowdown in the overall request allocation run time.

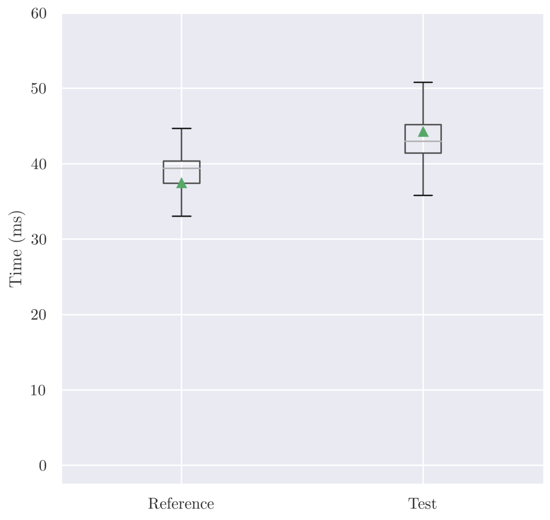


(a) Efficiency and stability scenario

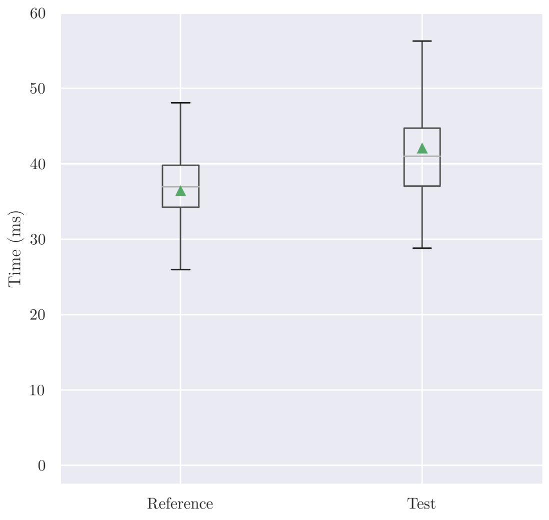


(b) Plasticity and convergence scenario

Figure 6.7: Average request allocation run time per episode



(a) Efficiency and stability scenario



(b) Plasticity and convergence scenario

Figure 6.8: Overview of the request allocation run time across all experiments

CHAPTER 7

Discussion

This chapter presents the findings of the research in relation to the research question stated in the introduction. The results observed from the experiments are discussed to form a conclusion. Furthermore, the implications of the extension are discussed, and possible future research is presented.

7.1 Comparison

The results of the experiments in both scenarios validate the extension of the CVI-Sim simulator. A slight deviation is observed across all performance measurements, apart from a more significant deviation of 7% in the average request allocation run time. The overall difference is expected since the Q-learning algorithm uses randomness to find the optimal policy (as described in Section 6.1.2).

The core metrics, violation and rejection ratio, serve as the key indicators for evaluating the test implementation. The similar observed behaviour as discussed in the results validates the extension. Similar patterns are observed within the other performance metrics of the experiments. The implementation shows near-identical behaviour in the server utilisation and the SFC non-collocation ratio. The server utilisation is maximised to roughly 90%, as anticipated by the server utilisation threshold (θ_l) (as described in Section 6.1). Crossing the server utilisation threshold leads to imposing a penalty on the agent. The similar behaviour in server utilisation confirms that the feedback is communicated correctly and can be acted upon in a similar manner as the reference implementation. In addition, the test implementation demonstrates a similar allocation strategy as the reference implementation, preferring to collocate VNFs of a service request on the same server. The identical strategy was expected since the implementations use the same mapping heuristic, striving to collocate VNFs.

The request allocation run time of the two implementations differs more significantly. The request allocation run time of the implementation is roughly 7% longer than the reference implementation, resulting in the simulations taking more time to complete. The slowdown was expected due to the introduction of the new communication layer. Although the experiments were performed on a single system, using only local networking, the overhead of ZeroMQ using a TCP connection between the processes has imposed a slowdown. A larger difference is seen in the first episode, this could be caused by the fact that the remote agent initialises the state of the model after receiving the first request, while the reference implementation does this before the experiment is started.

7.2 Conclusion

The CVI-Sim simulator has proven itself over the years as a performant and adequate framework for testing various resource allocation strategies. In recent research, machine learning, particularly reinforcement learning, has become a novel and promising approach to the SFC embedding problem. The machine learning community primarily uses the Python programming language with many actively developed libraries containing cutting edge research and low-level performance optimisations. The CVI-Sim simulator is written in Java, with a small and less vibrant machine learning ecosystem. This requires an algorithm developer to use low-level primitives, increasing bugs and slowing down research in the area. An extension was in need to allow for a loosely coupled integration between the simulator and a remote agent to provide access to the commonly used machine learning ecosystems.

The proposed extension provides a flexible and interoperable interface to the CVI-Sim simulator using a messaging system. A remote agent can use the interface to make admission control and resource allocation decisions on the substrate network of the simulator. In the research, the interface and a test implementation have been developed. The extension consists of a ZeroMQ socket acting as an interface to the simulator. The interface is integrated with the underlying simulator, allowing the agent to retrieve the substrate network state, as well as incoming service requests, and decide over the admission control (or potentially mapping of service requests). The messages sent over the socket are modelled using the Protobuf messaging format, providing low overhead and flexible integration in various programming languages.

The extension has been validated by comparing a test implementation to a reference implementation. The test implementation performs request admission control using the Q-learning algorithm. The implementation is based on a similar native Java-based implementation, employing the same Q-learning algorithm, part of the CVI-Sim project. The test implementation uses a remote agent to perform admission control, while the reference implementation entirely acts within the simulator. The extension has been validated by performing a set of identical experiments in different scenarios, using both implementations and comparing their results.

The results of the experiments confirm that the extension works correctly. The measurements of the test implementation show a replicated behaviour with a slight deviation within expectations due to the randomness used by the Q-learning algorithm to find the optimal policy. A small performance impact of 7% is measured in the average request allocation run time, induced by the overhead of the introduced communication layer.

The developed extension allows researchers to evaluate new algorithms. Development is no longer restricted to Java, allowing the design of algorithms in various programming languages. The separation eases the development and opens up new possibilities for experimentation, such as comparing algorithm implementations across different programming languages.

7.3 Future work

As explained in the comparison of the implementations, the current implementation suffers from a slower request allocation run time leading to longer simulation times. The newly introduced communication layer likely causes the slowdown. Researching the use of other network protocols could reduce the latency. The current implementation uses TCP for its known reliability, ease of use, and maturity. Research into alternatives has to take the reliability of the protocol into account to ensure correct simulations. As an example, the use of Unix domain sockets could be researched, which has a lower latency than TCP [21]. Using Unix domain sockets would impose a restriction on the operating system and require the simulator and remote agent to run on the same system.

The communication layer using ZeroMQ makes use of the PAIR socket type. The PAIR socket is restricted to one-to-one communication between two peers and allows unrestricted bidirectional communication. Future research could look at other socket types, allowing for more peers, providing more flexibility. Alternatively, the use of a separate socket per message flow could be researched; by doing so, stricter socket types can be configured, *e.g.,*, the request-reply socket type for the request message flow. This would improve reliability but increases the complexity and might have an slight performance impact.

Bibliography

[1] Edoardo Amaldi et al. “On the computational complexity of the virtual network embedding problem”. In: *Electronic Notes in Discrete Mathematics* 52 (June 2016), pp. 213–220. DOI: 10.1016/j.endm.2016.03.028.

[2] ZeroMQ Authors. *ZeroMQ*. https://zeromq.org/. [Online; accessed 14-April-2021]. 2007.

[3] Guruduth Banavar et al. “A Case for Message Oriented Middleware”. In: *Distributed Computing*. Vol. 1698. Sept. 1999, pp. 1–18. ISBN: 978-3-540-66531-1. DOI: 10.1007/3-540-48169-9_1.

[4] K. Bilal et al. “Trends and challenges in cloud datacenters”. In: *IEEE Cloud Computing* 1.1 (June 2014), pp. 10–20. ISSN: 2325-6095. DOI: 10.1109/MCC.2014.26.

[5] Rodrigo N. Calheiros et al. “CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms”. In: *Softw. Pract. Exper.* 41.1 (Jan. 2011), pp. 23–50. ISSN: 0038-0644. DOI: 10.1002/spe.995. URL: https://doi.org/10.1002/spe.995.

[6] Jin Cao et al. “Internet Traffic Tends Toward Poisson and Independent as the Load Increases”. In: *Nonlinear Estimation and Classification*. Ed. by David D. Denison et al. New York, NY: Springer New York, 2003, pp. 83–109. ISBN: 978-0-387-21579-2. DOI: 10.1007/978-0-387-21579-2_6. URL: https://doi.org/10.1007/978-0-387-21579-2_6.

[7] Danny De Vleeschauwer et al. *Reinforcement Learning for Service Function Chain Resource Allocation*. Discussion paper. Nokia Bell Labs, 2021.

[8] Philippe Dobbelaere and Kyumars Sheykh Esmaili. “Kafka versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper”. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*. DEBS ’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 227–238. ISBN: 9781450350655. DOI: 10.1145/3093742.3093908. URL: https://doi.org/10.1145/3093742.3093908.

[9] Abdur R. Fayjie et al. “Driverless Car: Autonomous Driving Using Deep Reinforcement Learning in Urban Environment”. In: *2018 15th International Conference on Ubiquitous Robots (UR)*. June 2018, pp. 896–901. DOI: 10.1109/URAI.2018.8441797.

[10] Xiaoyuan Fu et al. “Service Function Chain Embedding for NFV-Enabled IoT Based on Deep Reinforcement Learning”. In: *IEEE Communications Magazine* 57.11 (Nov. 2019), pp. 102–108. ISSN: 1558-1896. DOI: 10.1109/MCOM.001.1900097.

[11] GENI. *Resource Specification (RSpec) Documents in GENI*. https://groups.geni.net/geni/wiki/GENIExperimenter/RSpecs. [Online; accessed 12-April-2021]. 2012.

[12] Juliver Gil Herrera and Juan Felipe Botero. “Resource Allocation in NFV: A Comprehensive Survey”. In: *IEEE Transactions on Network and Service Management* 13.3 (Sept. 2016), pp. 518–532. ISSN: 1932-4537. DOI: 10.1109/TNSM.2016.2598420.

[13] Google. *Protocol Buffers (Protobuf)*. ”[Online; accessed 19-April-2021]”. July 2008. URL: https://developers.google.com/protocol-buffers/.

[14] D. Happ et al. "Meeting IoT platform requirements with open pub/sub solutions". In: *Annals of Telecommunications* 72 (Feb. 2017), pp. 41–52. ISSN: 1958-9395. DOI: 10.1007/s12243-016-0537-4. URL: https://doi.org/10.1007/s12243-016-0537-4.

[15] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[16] Péter Hegyi and József Varga. "Telco Cloud Simulator". In: *2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. Sept. 2019, pp. 1–7. DOI: 10.1109/CAMAD.2019.8858483.

[17] Péter Hegyi, Norbert Varga, and László Bokor. "An advanced telco cloud simulator and its usage on modelling multi-cloud and 5G multi-access environments". In: *2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. Feb. 2018, pp. 1–3. DOI: 10.1109/ICIN.2018.8401637.

[18] IBM. *CPLEX Optimizer*. https://www.ibm.com/analytics/cplex-optimizer. [Online; accessed 12-April-2021]. 2013.

[19] Vineet John and Xia Liu. "A Survey of Distributed Message Broker Queues". In: *CoRR* abs/1704.00411 (Apr. 2017). arXiv: 1704.00411. URL: http://arxiv.org/abs/1704.00411.

[20] Nicola Jones. "How to stop data centres from gobbling up the world's electricity". In: *Nature* 561 (Sept. 2018), pp. 163–166. DOI: 10.1038/d41586-018-06610-y.

[21] Kangho Kim et al. "Inter-Domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen". In: *Proceedings of the Fourth ACM SIGPLAN / SIGOPS International Conference on Virtual Execution Environments*. VEE '08. Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 11–20. ISBN: 9781595937964. DOI: 10.1145/1346256.1346259. URL: https://doi.org/10.1145/1346256.1346259.

[22] Dzmitry Kliazovich et al. "GreenCloud: A Packet-Level Simulator of Energy-Aware Cloud Computing Data Centers". In: *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*. Dec. 2010, pp. 1–5. DOI: 10.1109/GLOCOM.2010.5683561.

[23] Jay Kreps, Neha Narkhede, Jun Rao, et al. "Kafka: A distributed messaging system for log processing". In: 11 (2011), pp. 1–7.

[24] Yuxi Li. *Deep Reinforcement Learning: An Overview*. 2018. arXiv: 1701.07274 [cs.LG].

[25] Joshua Madadhain et al. "Analysis and Visualization of Network Data using JUNG". In: *JSS Journal of Statistical Software MMMMMM YYYY* VV (Nov. 2004).

[26] L Magnoni. "Modern Messaging for Distributed Sytems". In: *Journal of Physics: Conference Series* 608 (May 2015), p. 012038. DOI: 10.1088/1742-6596/608/1/012038. URL: https://doi.org/10.1088/1742-6596/608/1/012038.

[27] A. Medina et al. "BRITE: an approach to universal topology generation". In: *MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 346–353. DOI: 10.1109/MASCOT.2001.948886.

[28] R. Mijumbi et al. "Design and evaluation of learning algorithms for dynamic resource management in virtual networks". In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. May 2014, pp. 1–9. ISBN: 978-1-4799-0913-1. DOI: 10.1109/NOMS.2014.6838258.

[29] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (Feb. 2015), pp. 529–33. DOI: 10.1038/nature14236.

[30] Janine Morley, Kelly Widdicks, and Mike Hazas. "Digitalisation, energy and data demand: The impact of Internet traffic on overall and peak electricity consumption". In: *Energy Research & Social Science* 38 (Apr. 2018), pp. 128–137. DOI: 10.1016/j.erss.2018.01.018.

[31] nsnam. *Network Simulator 2 (ns-2)*. June 2008. URL: https://www.isi.edu/nsnam/ns/.

[32] OASIS. *Advanced Message Queuing Protocol (AMQP) Version 1.0.* Ed. by Robert Godfrey, David Ingham, and Rafael Schloming. 2012. URL: `http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf`.

[33] Andrew M. Odlyzko. "Internet traffic growth: sources and implications". In: *Optical Transmission Systems and Equipment for WDM Networking II.* Ed. by Benjamin B. Dingel et al. Vol. 5247. International Society for Optics and Photonics. SPIE, 2003, pp. 1–15. DOI: `10.1117/12.512942`. URL: `https://doi.org/10.1117/12.512942`.

[34] C. Papagianni et al. "On the optimal allocation of virtual resources in cloud computing networks". In: *IEEE Transactions on Computers* 62.6 (June 2013), pp. 1060–1071. ISSN: 1557-9956. DOI: `10.1109/TC.2013.31`.

[35] Chrysa Papagianni, Panagiotis Papadimitriou, and John Baras. "Rethinking Service Chain Embedding for Cellular Network Slicing". In: *2018 IFIP Networking Conference (IFIP Networking) and Workshops.* May 2018, pp. 1–9. DOI: `10.23919/IFIPNetworking.2018.8697029`.

[36] Chrysa Papagianni et al. "5Growth: AI-driven 5G for Automation in Vertical Industries". In: *2020 European Conference on Networks and Communications (EuCNC).* June 2020, pp. 17–22. DOI: `10.1109/EuCNC48522.2020.9200919`.

[37] PlanetLAB. *SFA Resource Specifications (RSpecs).* `https://web.archive.org/web/20171020121115/http://svn.planet-lab.org/wiki/SFAResourceSpecifications`. [Online; accessed 12-April-2021]. 2017.

[38] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (Jan. 2016), pp. 484–489. DOI: `10.1038/nature16961`.

[39] Utkal Sinha and Mayank Shekhar. "Comparison of Various Cloud Simulation tools available in Cloud Computing". In: *International Journal of Advance Research In Computer And Communication Engineering* 4 (Mar. 2015), pp. 171–176. DOI: `10.17148/IJARCCE.2015.4342`.

[40] Audie Sumaray and S. Kami Makki. "A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform". In: *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication.* ICUIMC '12. Kuala Lumpur, Malaysia: Association for Computing Machinery, 2012. ISBN: 9781450311724. DOI: `10.1145/2184751.2184810`. URL: `https://doi.org/10.1145/2184751.2184810`.

[41] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[42] Arryon D. Tijsma, Madalina M. Drugan, and Marco A. Wiering. "Comparing exploration strategies for Q-learning in random stochastic mazes". In: *2016 IEEE Symposium Series on Computational Intelligence (SSCI).* Dec. 2016, pp. 1–8. DOI: `10.1109/SSCI.2016.7849366`.

[43] Inc VMware. *RabbitMQ.* `https://www.rabbitmq.com/`. [Online; accessed 14-April-2021]. 2007.

[44] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3 (May 1992), pp. 279–292. ISSN: 1573-0565. DOI: `10.1007/BF00992698`. URL: `https://doi.org/10.1007/BF00992698`.

[45] Bo Yuan and Bangbang Ren. "Embedding the Minimum Cost SFC with End-to-End Delay Constraint". In: *2020 5th International Conference on Mechanical, Control and Computer Engineering (ICMCCE).* Dec. 2020, pp. 2299–2303. DOI: `10.1109/ICMCCE51767.2020.00497`.

APPENDIX A

Protobuf message specification

```
syntax = "proto3";

message Parameter {
  string key = 1;
  string value = 2;
}

message NodeAllocation {
  int32 sourceId = 1;
  int32 destinationId = 2;
}

message SubstrateLinkAllocation {
  int32 id = 1;
}

message RequestLinkAllocation {
  int32 id = 1;
  repeated SubstrateLinkAllocation allocations = 2;
}

message RequestAllocation {
  string requestId = 1;
  bool success = 2;
  repeated NodeAllocation nodeAllocations = 3;
  repeated RequestLinkAllocation linkAllocations = 4;
}

message Action {
  int32 id = 1;
  repeated RequestAllocation requestAllocations = 2;
}

message Node {
  int32 id = 1;
  repeated Parameter parameters = 2;
}
```

```
message Link {
  int32 id = 1;
  int32 sourceNodeId = 2;
  int32 destinationNodeId = 3;
  repeated Parameter parameters = 4;
}

message ServiceRequest {
  string id = 1;
  repeated Node nodes = 2;
  repeated Link links = 3;
}

message Substrate {
  string id = 1;
  repeated Node nodes = 2;
  repeated Link links = 3;
}

message Request {
  repeated Substrate substrates = 1;
  repeated ServiceRequest requests = 2;
}

message Feedback {
  double value = 1;
}
```

APPENDIX B

Request message example

B.1 Request

In the following Listing B.1, the example SFC and substrate network from Figure 2.2a and Figure 2.2b have been modelled as a `Request` message using JSON. Although, Protobuf is used in the implementation, JSON can visualise the data since Protobuf uses a binary format (unreadable to the human eye). The example message is sent from the simulator to the remote agent, requesting for an allocation.

Listing B.1: `Request` message example in JSON

```
{
  "substrates": [
    {
      "id": "sub0",
      "nodes": [
        {"id": 0, "parameters": [{"cpu":  3, "available_cpu": 3}]}, // A
        {"id": 1, "parameters": [{"cpu":  3, "available_cpu": 3}]}, // B
        {"id": 2, "parameters": [{"cpu":  2, "available_cpu": 2}]}, // C
        {"id": 3, "parameters": [{"cpu":  5, "available_cpu": 5}]}, // D
        {"id": 4, "parameters": [{"cpu":  3, "available_cpu": 3}]}, // E
        {"id": 5, "parameters": [{"cpu":  4, "available_cpu": 4}]}  // F
      ],
      "links": [
        {"id": 0, "from": 0, "to": 1,"parameters":
            [{"bandwidth":  5, "available_bandwidth": 5}]}, // A — B
        {"id": 1, "from": 1, "to": 2,"parameters":
            [{"bandwidth":  3, "available_bandwidth": 3}]}, // B — C
        {"id": 2, "from": 2, "to": 5,"parameters":
            [{"bandwidth":  2, "available_bandwidth": 2}]}, // C — F
        {"id": 3, "from": 2, "to": 4,"parameters":
            [{"bandwidth":  2, "available_bandwidth": 2}]}, // C — E
        {"id": 4, "from": 0, "to": 3,"parameters":
            [{"bandwidth":  5, "available_bandwidth": 5}]}, // A — D
        {"id": 5, "from": 3, "to": 4,"parameters":
            [{"bandwidth":  3, "available_bandwidth": 3}]}  // D — E
      ]
    }
  ],


  "requests": [
    {
      "id": "req42",
      "nodes": [
        {"id": 0, "parameters": [{"cpu":  3}]}, // x
        {"id": 1, "parameters": [{"cpu":  4}]}, // y
        {"id": 2, "parameters": [{"cpu":  2}]}  // z
      ],
      "links": [
        {"id": 0, "from": 0, "to": 1,"parameters":
            [{"bandwidth":  5}]}, // x --> y
        {"id": 1, "from": 1, "to": 2,"parameters":
            [{"bandwidth":  2}]}  // y --> z
      ]
    }
  ]
}
```

B.2 Action

Listing B.2 shows an example `Action` message returned by the remote agent. The message has been modelled in the context of the `Request` message, as shown in the previous section. The `Action` message contains the embedding as shown in Figure 2.2c.

Listing B.2: `Action` message example in JSON

```
{
  "requestAllocations": [
    {
      "requestId": "req42",
      "success": true,
      "nodeAllocations": [
        {"id": 0, "sourceId": 0, "destinationId": 0}, // x --> A
        {"id": 0, "sourceId": 1, "destinationId": 3}, // y --> D
        {"id": 0, "sourceId": 2, "destinationId": 4}, // z --> E
      ],
      "linkAllocations": [
        {"id": 0, "allocations": [4]}, // (x --> y) --> (A -- D)
        {"id": 1, "allocations": [5]}, // (y --> z) --> (D -- E)
      ]
    }
  ]
}
```