Multi-Agent Software Defined Network controller: An approach to disaggregate the control plane

A thesis submitted by **Vijaya Durga Chemalamarri**

in fulfilment of the requirements for the award of the degree

Doctor of Philosophy

in Engineering

Faculty of Engineering and Information Technology

University of Technology Sydney

April 2023

Certificate of Original Authorship

I, Vijaya Durga Chemalamarri, declare that this thesis is submitted in fulfilment of the requirements for the award of Doctor of Philosophy, in the Faculty of Engineering and Information Technology at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.

This research is supported by the Australian Government Research Training Program.

Signature: Production Note:
Signature removed prior to publication.

Date: 18-April-2023

Acknowledgement

I acknowledge the guidance provided by my primary supervisor Prof. Robin Braun. Prof. Robin's patient, calm and timely guidance helped alleviate and address challenges that arose during the course of my Ph.D. I acknowledge the guidance provided by my co-supervisor, Prof. Mehran Abolhasan. Prof. Mehran always encouraged me to aim higher and provided guidance and feedback at various stages of my Ph.D. I am immensely thankful to them for their mentorship. I also acknowledge editing services provided by John Hazelton, thanks to whom, this thesis is readable. I also would like to mention my friend, Rahul Anand for our discussions on computer science and programming. Sometimes, these discussions gave me a different insight of the problem in hand. I thank all the constructive feedback provided by reviewers for my published work. I must not forget to acknowledge the makers of Python programming language and makers of Asyncio, Python-OpenFlow, ZMQ, and OsBrain Python libraries. I also would like to thank the makers of StackOverFlow.

My husband, Deba had been a rock-solid support during the entire duration of my Ph.D. I could really not have gone through the process, if not for his support, encouragement and optimism during my lows and there were many such lows. My parents have always taught my sister and me to keep aiming higher and work towards achieving and fulfilling goals. This work is a tribute to their lesson. I also thank my sister for always setting a very high bar for me to achieve. My in-laws were as serene as ever during my study and I thank them for that. I thank my friends whose impromptu meetings helped ease off some stress.

...and my cats for providing healthy dose of daily dopamine.

Abstract

This thesis proposes a modular Software Defined Network controller - (Multi-agent SDN controller). The thesis is in two parts. The first part makes the proposition and provides the necessary background information. The Software Defined Network paradigm deviates from traditional networks by logically centralising and physically separating the control plane from the data plane. Currently, SDN control planes are monolithic, OpenFlow dependent and do not support application portability. A modular architecture is proposed in the second part of the thesis to improve the existing control plane. The architecture comprises social agents. The architecture varies from reactive to learning agents based on the agent's role and a knowledge base to formally represent knowledge in a multi-agent system and provide simple inferences. For information exchange, the agents exchange messages. A prototype is designed and built to demonstrate controller operation in some common network scenarios. Performance is evaluated at both the agent and system levels. While it was observed that agent system performance compares well with a monolithic controller, the agent system has higher latency when physically distributed.

Publications Supporting this Thesis

The following is a list of accepted publications resulting from this thesis.

Conferences

1. Chemalamarri, V. D., Braun, R., Lipman, J., Abolhasan, M. (2018, November). A Multi-agent Controller to enable Cognition in Software Defined Networks. In 2018 28th International Telecommunication Networks and Applications Conference (ITNAC) (pp. 1-5). IEEE.

2. Chemalamarri, V. D., Braun, R., Abolhasan, M. (2020, November). Constraint-Based Rerouting mechanism to address Congestion in Software Defined Networks. In 2020 30th International Telecommunication Networks and Applications Conference (ITNAC) (pp. 1-6). IEEE.

3. V. D. Chemalamarri, M. Abolhasan and R. Braun, "An agent-based approach to disintegrate and modularise Software Defined Networks controller," 2022 IEEE 47th Conference on Local Computer Networks (LCN), 2022, pp. 407-413, doi: 10.1109/LCN53696.2022.9843585

It is anticipated that the research outcomes are to be published in a peer reviewed journal. Information about papers that are being prepared but remain to be published should be included here.

Contents

List of Tables

List of Figures

Part I

Background & Proposition

Chapter 1

Proposal

1.1 Introduction

This thesis sets out to build a modular Software Defined Networks control plane. The control plane comprises autonomous, rational and social agents capable of sensing, learning, reasoning and communicating. The agents employ learning techniques such as reinforcement learning and incremental learning and reasoning mechanisms such as constraint solving and inference to deduce unknown facts. The agents use local and global knowledge base to store network information while exchanging messages to share local information.

Contributions of this thesis include

1. an architecture to disintegrate the SDN control plane
2. a modular multi-agent based SDN control plane
3. an ontology and knowledge base for the agent system
4. agent interaction protocols

The above contributions culminate towards a prototype for a modular and physically distributed SDN controller. The proposed modular controller can function as a one

control entity on both physically localised (a single node) and physically distributed (multiple nodes, cloud) infrastructure.

1.2 Motivation and Proposal

In this section the motivation for the thesis and proposition to build an agent-based controller are presented.

1.2.1 Motivation

Traditional networks are distributed by design, consisting of devices that perform dedicated operations leading to closed, propriety networks. The Software Defined Network (SDN) paradigm deviates from traditional networks by logically centralising and physically separating the control plane from the data plane. The control plane communicates with the data plane via a Southbound Interface (SBI). While the control plane instructs the data plane on packet handling, switches store the instructions in flow tables as flow rules. Flow rules match against all incoming packets and execute corresponding actions. In case of no match, the switch forwards the packet to the controller for processing. Northbound Applications (NBA) running on the controller process the incoming packet, create a flow rule and configure this rule on the switch. Northbound Interfaces (NBI) establish a communication channel between the controller core module and applications. Most SDN controllers fall into one of three categories a) physically centralised, b) physically distributed, logically centralised, and c)physically distributed, logically distributed. Physically centralised controllers suffer from a single point of failure. Physically distributed controllers require synchronisation for information exchange. Other undesirable features of most of the existing control planes are listed below:

1. *SBI Dependent:* SDN is synonymous with OpenFlow (OF) [1]. Such an association is beneficial for standardising SBIs; nevertheless, it results in severe

dependency on a single SBI and hinders innovation. SDN applications and OpenFlow are heavily intertwined. Most existing networks are traditional (non-SDN), and adopting SDN is gradual. Such a setup with SDN and non-SDN devices would result in a hybrid network. OpenFlow is incapable of managing traditional networks. The absence of a singular SBI to manage hybrid networks consisting of various forwarding devices (both SDN data paths and traditional network devices (wired and wireless)) leads to physically and logically distributed control planes. By decoupling applications from OpenFlow, alternative applications and SBIs may be explored independently for a logically centralised control plane for hybrid networks.

2. *Monolithic:* Monolithic software is a self-contained software application. Components in monolithic software are tightly interdependent, coupled with one another and act as a single entity [2]. Many smaller applications benefit from monolithic design. They are easy to deploy and maintain and perform better. Nevertheless, monolithic applications are rigid with resource allocation and hard to scale and manage. Such applications also require extensive testing for minor and major upgrades. Unfortunately, most of the existing SDN controllers are monolithic by design [3], [4], [5]. NBAs, controller core and OpenFlow protocol are tightly coupled. Monolithic design is not restricted to physically centralised controllers. Distributed control plane consists of multiple such replicas of the controller instances and do not necessarily modularise the controller.

3. *Application portability and code redundancy:* Consider POX [6] and Ryu [7] controllers. Both controllers are python-based SDN controllers; nevertheless, applications written for the POX controller cannot run on the Ryu controller and vice versa. Poor application portability is due to the non-standardisation of NBIs. Current NBAs rely heavily on the interface provided by the controller core. Most SDN applications perform a standard set of actions such as exchanging packets with the data plane, parsing and un-parsing OF packets

and storing network information creating redundant code; consolidating such essential functions increases code re-usability.

4. *Intelligence:* Current SDN applications are reactive and rule-based, where a domain expert defines their functionality. SDN provides a centralised view of the environment opening up opportunities to build intelligent applications capable of reacting to environmental changes by sensing, reasoning, acting and learning.

This thesis aims to address above mentioned issues of existing SDN control plane by setting and achieving the goals illustrated in Figure 1.1 and described in Section 1.2.2.

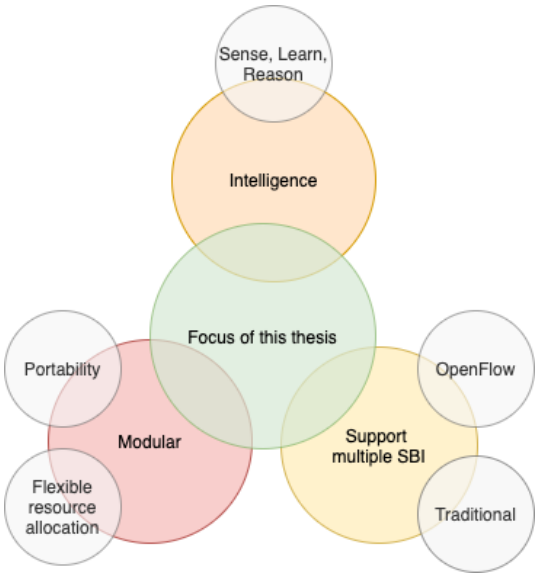


Figure 1.1: Focus of this thesis

1.2.2 Proposal

To address the issues listed in Section 1.2, this thesis proposes to build an SDN controller that has the following characteristics:

1. *Modular:* Modular design is an alternate approach to monolithic architecture. Modular systems are built by disintegrating monolithic applications into components (e.g. micro-services or agents). Modular architecture allows isolation and flexible allocation of resources such as CPU and memory to components. Components in such distributed systems are social and exchange information with other components.

2. *Supports one or more SBI:* A logically centralised control plane allows seamless information exchange between traditional and OpenFlow devices in a network. Though OpenFlow is the desired SBI to manage SDN devices, the control plane also receives, processes and uses information acquired from other managing protocols such as NETCONF [8], SNMP [9].

3. *Intelligent:* Controller intelligence resides at multiple levels (application level and system level). Application level intelligence combines SDN's centralised view of the data plane with learning and reasoning techniques. System level intelligence is attained by communicating and coordinating applications.

The proposed architecture is called Multi-Agent SDN (MASDN). MASDN is a SDN control plane architecture that consists of multiple social and rational agents. The agent system is modular, supports multiple SBIs and possess application-level intelligence. A prototype is developed to evaluate the operation and performance of the proposed architecture.

Distributed systems comprise a group of entities. These entities are micro-services or agents. Unlike micro-services, agents can perceive, measure and act in an environment. They are autonomous and self-sufficient. From an implementation

perspective, each agent runs as a system process, allowing isolated resource allocation and fault detection.

Since agents are autonomous, various deployment options can be explored, such as physically centralised, physically distributed, and cloud-based deployments. Each agent in the agent system performs a singular network task. Agent system is scalable to include additional agents. Agent upgrade and testing is isolated to a specific agent, reducing efforts to upgrade and test the entire agent system. For example, protocols such as NETCONF [8] can be wrapped into agents and added to the agent system. With the assistance of a global knowledge base, all SBIs can manage and maintain a hybrid network realising a centralised control plane for hybrid networks.

The autonomous and social nature of agents also makes applications portable. All agents need not adopt the same technology stack to operate seamlessly. The agent system can be deployed centrally on a single physical machine or distributed across multiple physical devices (physical and virtual) while using a shared knowledge base to maintain consistent information across various agents.

Social agents open up the possibility of composing applications differently, though such composition is different from the goals this thesis aims to meet. Consider two agents (Ag1 and Ag2). Ag1 performs load balancing, and Ag2 ensures QoS compliance. In a traditional SDN setup, these agents are composed serially or in parallel to yield a single flow rule. The proposed agent system facilitates the agents to communicate, negotiate and arrive at a single composed flow rule. This thesis layouts the foundation for such interaction but does not propose any specific framework for negotiation and flow composition though such negotiations are possible in future.

1.3 Key contributions

This work presents a modular, agent-based SDN controller architecture (MASDN) as a step toward building an intelligent SDN control plane. The control plane comprises intelligent, social and autonomous agents capable of interacting with the

environment (including other agents) reasoning, learning and acting. This thesis makes the following contributions:

1. *Modular control plane* architecture comprising of below agents
   (a) *SBI agents* to interact with the data plane.
   (b) *IP path agent* to compute end-to-end path for data exchange.
   (c) *Reroute agent* reroutes flows while ensuring congestion does not transfer to other network parts. The agent reasons flow and path assignment as a constraint satisfaction problem to find the optimal path to reroute a flow.
   (d) *TCP fairness agent* ensures TCP fairness amongst flows that do not share the same start time (in other words, non-synchronised flows) by altering the congestion window of TCP flows. This agent uses reinforcement learning to learn the desired behaviour.
   (e) *Incremental learning-based traffic prediction agent* employs incremental learning to predict network traffic.
   (f) *Topology agent* periodically discovers topology.
   (g) *Port status agent* handles port failures and reroutes traffic.
   (h) *Port stats agent* gathers port statistics. These statistics are used by other agents, such as rerouting and traffic prediction agents.
   (i) *Flow table agent* to remove stale flow rules from global knowledge base.
   (j) *Knowledge Base agent* to manage the knowledge base.
2. *Ontology and knowledge base* is also presented for use by agent systems. The knowledge base agent executes a reasoner on the ontology to ensure consistency and detect network abnormalities such as link failures and congestion. The knowledge base agent notifies management agents of the abnormality for further action.
3. *Interaction protocols* that agents use to communicate and exchange messages.

1.4 Methodology

This thesis adopts *build* methodology to prototype the proposed multi-agent system. [10] defines *build* methodology "as the process of building an artifact (software or hardware) system to demonstrate an idea". Build approach [10] provides guidelines for building a prototype to demonstrate an idea. These guidelines include - planning the approach to prototype before building, reusing existing components, choosing a programming language and periodic testing.

Multiple methodologies have been proposed to decompose a monolithic system into a multi-agent one. Methodologies such as GAIA [11], Tropos [12], MESSAGE [13], Prometheus [14], MaSE [15], and O-MaSE [16] are a few examples. From a system level, the SDN controller performs periodic actions such as monitoring, algorithm-based tasks such as path-computation for provisioning, reasoning tasks such as maintaining a consistent network state and other advanced tasks involving learning. Different agents perform different tasks. Thus, an agent's architecture is determined by the task an agent performs, resulting in agents that are simple reactive to proactive learning agents. This thesis adopts MaSE methodology to decompose the SDN controller into a multi-agent system. MaSE methodology is goal-based, independent of agent architectures, programming language or communication framework. MaSE supports UML representation and enables the building of heterogeneous agent systems comprising agents with different architectures ranging from simple reactive agents to complex learning agents. A simplified waterfall version of the methodology is presented in Table 1.1.

Finally, the ontology presented in the thesis is used by the agents to build messages for communication. Relationships are established in the ontology to build a knowledge base. The knowledge base agent performs basic inferences on the knowledge base to identify inconsistencies. Some common network use cases are tested in two deployment methods - on a single node (physically centralised), on multiple physical nodes (distributed).

State	Activities	Tasks
Requirement Analysis	Capture Goals	Model Goals Generate goal hierarchy
	Apply Use Cases	Use Cases Sequence Diagrams
	Refining Roles	Define Roles Concurrent Tasks
Design	Create Agent Classes	Agent Classes
	Construct Conversations	Conversations
	Assemble Agent Classes	Agent Architecture
	System Design	Deployment Diagrams

Table 1.1: MaSE Methodology [15]

1.5 Results

The prototype of the proposed modular multi-agent SDN controller was evaluated using various use case scenarios, including centralised, distributed, and agent and system level. At the agent level, the operations of three complex agents are evaluated. System level performance is evaluated in both operational modes and compared against monolithic controller- Ryu [7]. The comparison is in terms of flow setup duration and bandwidth conducted in network scenarios.

1.6 Outline

This thesis is organised into two parts: proposition and contribution. Part 1 of the thesis presents the proposal (Chapter 1) and discusses background information (Chapter 2, Chapter 3).

Chapter 2 introduces the reader to the field of Software Defined Networks (SDN). This chapter describes the idea of separation of the data plane from the control plane, discusses multiple control plane architectures supported by a discussion on relevant literature.

Chapter 3 introduces the field of multi-agent systems along with the concepts of learning and reasoning in the agents. This chapter also briefly touches on knowledge representation and presents a brief discussion on existing agent communication languages.

Part - II (Chapter 4, Chapter 5, Chapter 6) of this thesis focuses on contributions.

Chapter 4 presents multi-agent SDN controller architecture by breaks down the monolithic controller to an agent system. This chapter then proceeds to present the proposed modular architecture and describes individual agents' operations in detail.

Chapter 5 describes knowledge representation, presents an ontology, lists and discusses the inferences made on the knowledge. To enable information exchange among agents of the agent system, messages and communication patterns are presented in the later part of the chapter.

Chapter 6 describes the framework used and presents the prototype. Various operational modes of the prototype are presented and the prototype's operation is tested in multiple network scenarios. Finally, results for the fore-mentioned tests are presented followed by a discussion.

Chapter 7 concludes this thesis and discusses some improvements and scope for future work.

Chapter 2

Software Defined Network

2.1 Introduction

Network infrastructure comprises forwarding devices, security devices and other monitoring and management entities. Most forwarding devices, such as switches and routers, have a data plane, a control plane, and a management plane to configure and control data and the control plane. Ports and forwarding tables form the data plane of a switch. The data plane forwards the packets based on information in the forwarding table. In an instance when a forwarding device cannot handle an incoming packet, the packet is handled by the control plane. The control plane runs control protocols, updates the forwarding tables and enables communication with other devices in the network.

A control plane is a central processing unit for handling generic device-directed functions and implements control protocols such as Spanning Tree Protocol (STP), Open Shortest Path First (OSPF), Border Gateway Protocol (BGP), and Local Link Discovery Protocol (LLDP) and Internet Group Management Protocol (IGMP), amongst others. On the other hand, the data plane has an Application-Specific Integrated Circuit (ASIC) designed to perform specific functions configured in its pipeline. ASICs are function specific and use Content-Addressable Memory (CAM) and Tertiary CAM (TCAM) to store forwarding information, Access Control Lists

(ACLs) and Quality of Service (QoS) policies. A very high-speed lookup on the CAM table leads to line-rate transmission speeds. ASIC pipeline design is vendor-specific, leading to vendor lock-ins. Each network device operates as a fully autonomous entity and conveys its existence to other devices by exchanging information with other devices that run similar communication protocols. Exchanged information includes routing tables, multi-cast membership, and port configuration information. This information does not include data plane configurations such as QoS policies or ACLs. Since the configuration and management of these protocols is a manual task, human errors are possible, leading to inconsistency in forwarding information across the network. Figure 2.1 reproduces an abstract, modified view of the operation of traditional network devices presented in [17].

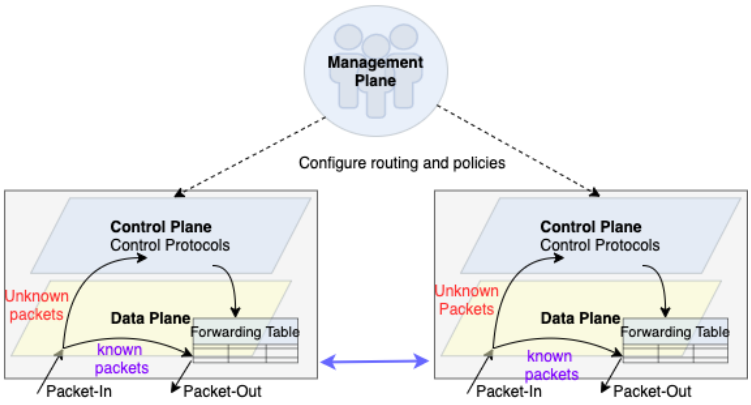


Figure 2.1: Traditional network with two interconnected devices

The ASIC makes Ethernet switches perform at significantly faster rates to achieve higher throughput while inherently making the device cater to only one application, quickly adding to a high number of application-specific devices [17]. SDN aims to simplify the control plane while making way for vendor-agnostic devices. SDN advocates decoupling control and data plane by physically separating and logically aggregating the control plane from the data plane of a forwarding device such as a switch. This separation effectively leads to a central control entity that manages the data plane. A centralised control plane generates a consistent topology, QoS, ACL policies. The control plane uses this universal network information and composes forwarding policies for the data plane. Centralisation of control also leads to building

more advanced network services and functions [18]. Applications running on the control plane process the incoming packet using application logic. For example, a firewall application can determine whether a packet is safe for admission. The Control plane translates these results into flow rules and inserts them into the switches in the data plane. This updated network architecture is shown in Figure 2.2.

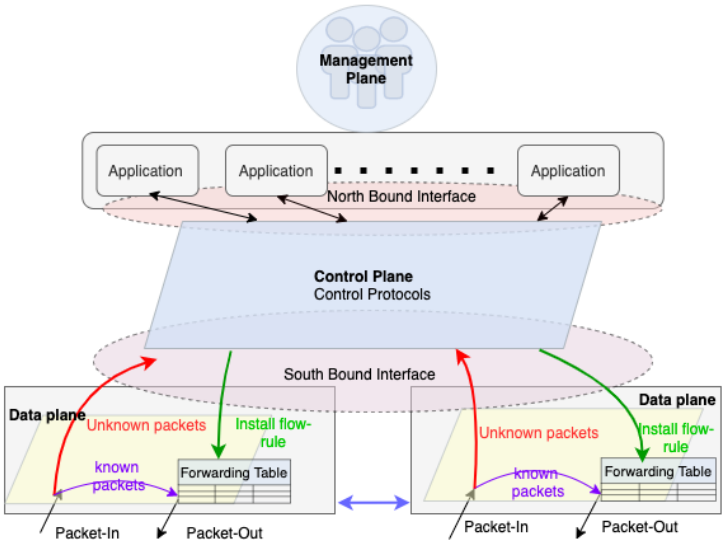


Figure 2.2: A Software Defined Network with decoupled data and control plane

2.2 Software Defined Network

This section provides a quick primer on Software Defined Network (SDN) architecture. Drawing parallels from a traditional network, SDN also has a data plane, a control plane and an applications/ management plane. Unlike traditional networks, the control plane of SDN is logically centralised and programmable, and the data plane consists of interconnected forwarding elements (also known as data paths). A set of applications running on the control plane execute network functions. Some examples are path-finding, firewall, load balancing, resource management, policy enforcement, authorisation and authentication applications. Southbound Interface (SBI) enables the communication between the control and data planes, and a Northbound Interface (NBI) enables the communication between the control plane and the application plane. An east-west bound communication interface facilitates inter-controller

communication. A summary on each plane is presented as follows.

2.2.1 Data Plane

Data Plane comprises (a) inter-connected forwarding elements (FE) and (b) SBI protocol.

2.2.1.1 Forwarding Element

A Forwarding Element (FE) is a hardware or software data plane entity that forwards and filters packets. A *hardware* FE is a physical switch with a fixed-function switch ASIC or a programmable switch ASIC or Field-Programmable Gate Array (FPGA). FE receives, parses the incoming packet, identifies header fields and matches the fields against match-action rules. In a fixed-function switch ASIC, each stage matches a specific header field and thus limits the number of protocols the switch can match against. The switch hardware needs a replacement to support matching against new protocol headers.

On the other hand, in a Programmable ASIC, each stage is programmable; that is- it is possible to program a stage to match against a previously unmatched protocol header. Such programmable ASICs allow flexible match-action forwarding logic. Protocol Independent Switch Architecture (PISA) forms the underlying architecture for a programmable ASIC [19]. Programming Protocol-Independent Packet Processors (P4) [20] programming language is used to program PISA. Using P4 programming language a programmer can define custom headers to match, actions to take and exact processing sequence. P4 compiler compiles the program, produces a target-specific configuration binary to configure the underlying ASIC.

A *software* FE is a virtual switch operating on any general-purpose processor. Unlike ASICs in hardware switches, software switches are CPU based, making them operate relatively slower than ASIC-based hardware devices. Software switch components include virtual NIC, ports to send and receive packets, kernel to process

packets and data structures to realise flow tables. Unlike fixed-size TCAMs used in hardware switches, software implementation of flow-table circumvents size limitation [17].

OpenVSwitch (OVS) [21] is a widely adopted virtual switch that supports OpenFlow. OVS switch has a *kernel* module for fast and efficient packet handling. *ovs-vswitchd* manages multiple ovs instances and handles packets sent by the kernel module. Usually, the first packet of the flow is sent to ovs-vswitchd. Ovs-vswitchd communicates with *ovsdb-server* and external controller. While ovsdb stores switch configuration, the external controller provides forwarding information to ovs-vswitchd. The flow table is updated with this information (known as the flow rule). All subsequent packets are handled according to this flow rule.

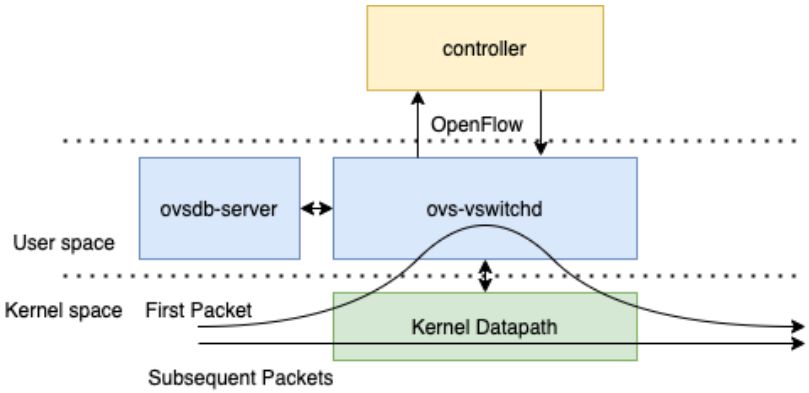


Figure 2.3: OpenVSwitch [21]

A FE are also referred as a *datapath* and this thesis uses both the terms as synonymous.

2.2.1.2 OpenFlow

Southbound Interface (SBI) enables the communication between the control and data planes. The most popular SBI is OpenFlow [1]. OpenFlow was developed primarily for network experimentation aimed toward building programmable networks. Open Network Foundation (ONF) [22] standardised OpenFlow pipeline and protocol specifications [23].

The OpenFlow pipeline consists of multiple forwarding tables. Each table is composed of multiple flow rules. A flow rule is an *action* that a switch performs upon *matching* against a packet. OpenFlow pipeline processing is the order of matching an incoming packet against flow tables. OpenFlow pipeline is an abstraction mapped to the switch's actual hardware as noted in [23], shown in Figure 2.4.

OpenFlow-enabled switches can either be OpenFlow-hybrid or OpenFlow-only [23]. OpenFlow-only switches handle packets using only the OpenFlow pipeline, while OpenFlow-hybrid switches can process packets either using the OpenFlow pipeline or the non-OpenFlow (L2-switch) pipeline.

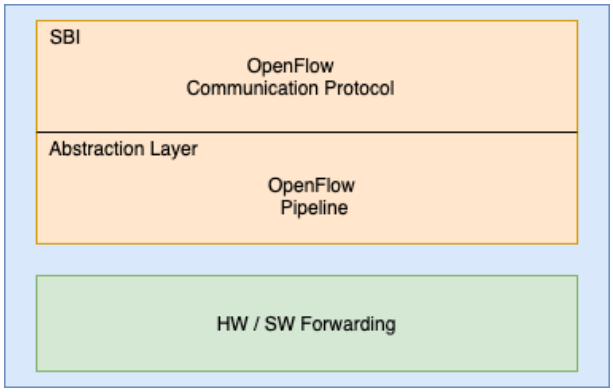


Figure 2.4: OpenFlow switch

Flow tables store flow rules. Each flow rule has match criteria, action criteria and counters. A match is a list of all protocol header fields of an incoming packet the flow rule will match against, and instructions are a list of actions mapped against the fore-mentioned match criteria. Match fields and action fields are listed in Figure 2.5. A counter associated with a flow rule maintains statistics for the rule. The number of transmitted bytes, received bytes and number of packets processed by the queue are maintained using counters per flow rule. Apart from the flow table, the group table and meters table entries also maintain counters.

OpenFlow packet processing pipeline starts at table 0. The switch performs lookups on the flow table, matching incoming packet headers against flow rules, incoming ports and metadata of the packet. Upon a successful match, the switch executes corresponding actions. Actions can include, updating counters, headers,

Match Protocol	
IN_PORT	IPV4_SRC
ETH_DST	IPV4_DST
ETH_SRC	IP_PROTO
ETH_TYPE	ICMPV4_TYPE
ARP_OP	ICMPV4_CODE
ARP_SPA	TCP_SRC
ARP_TPA	TCP_DST
ARP_SHA	UDP_SRC
ARP_THA	UDP_DST

(a) Select Match feilds

Mandatory/Optional	Action	Description
Mandatory	Output	Output to switch port
Optional	Set VLAN IP	Set the IEEE802.1q VLANID
Optional	Set VLAN PCP	Set IEEE802.1q priority
Optional	Strip VLAN	Strip the IEEE802.1q header
Optional	Set Ethernet source address	Set Ethernet Source address
Optional	Set Ethernet destination address	Set Ethernet Destination address
Optional	Set IP source address	Set IP source address
Optional	Set IP destination address	Set IP destination address
Optional	Set IP TOS	Set IP Type of Service(ToS) bits
Optional	Set TCP/UDP source port	Set TCP/UDP source port
Optional	Set TCP/UDP destination Port	Set TCP/UDP destination port
Optional	Enqueue	Output packet to queue

- Physical ports
- Virtual Ports
  - All
  - Controller
  - Local
  - Table
  - in_port
  - Normal
  - Flood

If no action is listed, default action is to '**drop**' the packet

(b) Actions

Figure 2.5: Match, Action field values of a flow rule

sending to a specific outport or sending to next table for processing. If a lookup does not return an action, that is, if a packet does not match against any flow rule, the switch performs actions against a table-miss flow rule. In the absence of table-miss entry, datapath drops the packet. Figure 2.6 reproduces the pipeline from [23].

OpenFlow Protocol standardises communication patterns and messages between the control and data plane. Some exchanges are initiated by the controller, e.g., to install a flow rule in the flow table, and some by the switch, e.g., to send information about dead ports to the controller. As mentioned below, three communication patterns exist between the controller and the data plane.

- **Symmetric** messages are initiated by either the switch or controller.

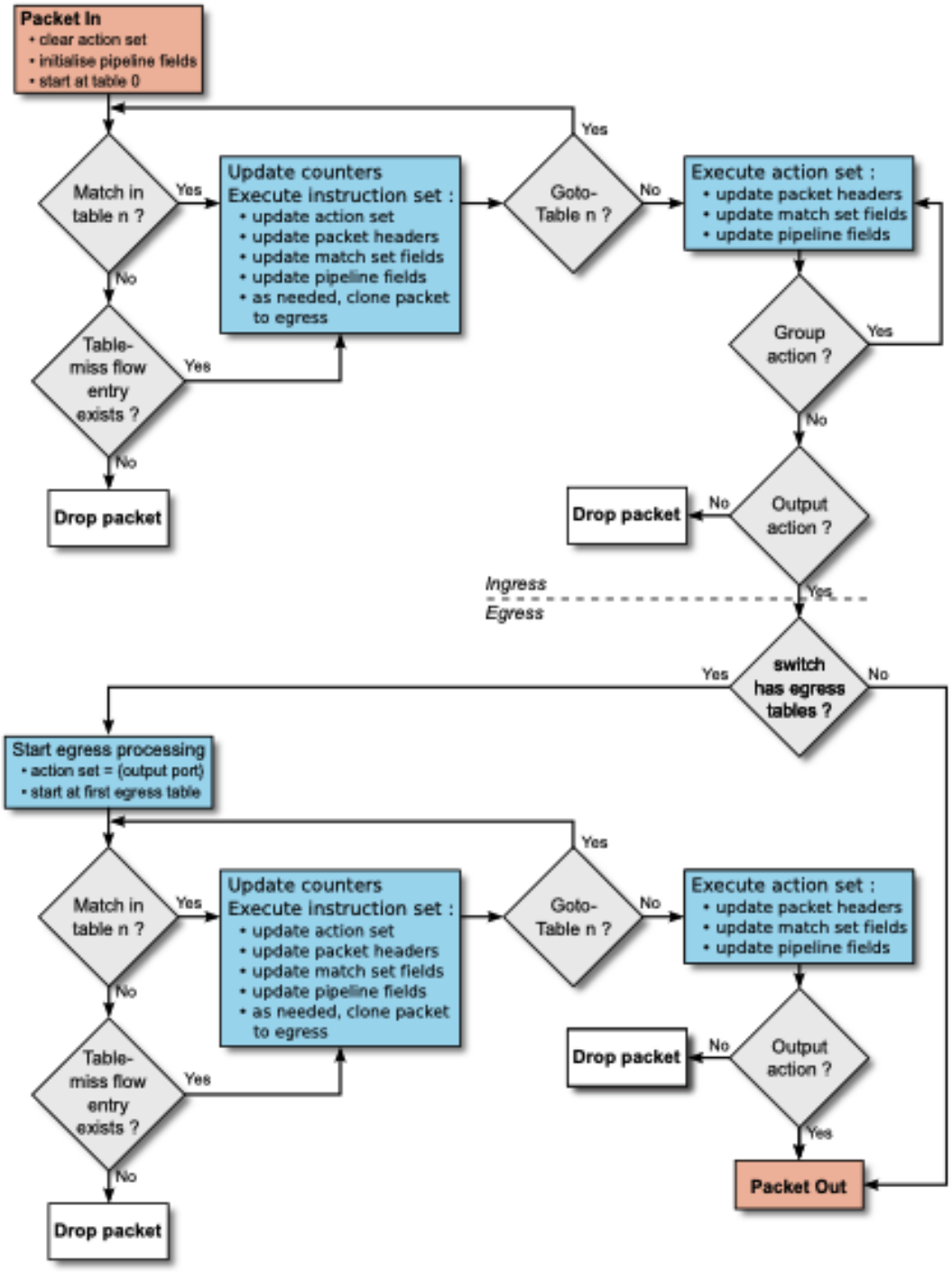


Figure 2.6: OpenFlow Pipeline processing [23]

OFPT_HELLO, OFPT_ECHO_REQUEST and OFPT_ERROR_MSG messages are an example of symmetrical messages. OFPT_HELLO messages are exchanged during the connection startup phase, and echo messages are exchanged to keep the connection alive. Error messages are exchanged to notify each other of connection problems.

- **Asynchronous** messages are unsolicited messages a switch sends to the controller to inform the controller about a state change. The switch sends an OFPT_PACKET_IN message to pass the control of a table-miss packet to the control plane. An OFPT_FLOW_REMOVED message notifies the controller of a removed flow rule. OFPT_PORT_STATUS message is used to notify the controller about any port state changes. OFPT_ROLE_STATUS, OFPT_TABLE_STATUS and OFPT_REQUEST_FORWARD are other asynchronous messages sent by the switch.

- **Controller-to-switch** messages are sent by the controller intending to gather switch information or change the switch's state. These messages include OFPT_FEATURES_REQUEST and OFPT_GET_CONFIG_REQUEST. The controller uses OFPT_FEATURES_REQUEST to query switches' capabilities during the handshake process. The controller modifies the behaviour of the flow table using an OFPT_TABLE_MOD message, and flow rules within the flow table are modified using the OFPT_FLOW_MOD message. Adding a new rule, modifying an existing rule and deleting a rule are considered modifications to a flow rule. The physical port's behaviour is modified using the OFPT_PORT_MOD message. Group tables are modified using the OFPT_GROUP_MOD message, and OFPT_METER_MOD are used to modify meter tables.

  The controller also initiates OFPT_MULTIPART_REQUEST, a multipart request message. The multipart message type field distinguishes various multipart messages. The controller sends a OFPMP_DESC message to gather information about the switch. While a OFPMP_FLOW message is sent to collect individual flow stats, OFPMP_PORT_STATS is sent to gather port statistics. OFPMP_QUEUE_STATS message is used to collects Queue statistics. The controller uses OFPT_PACKET_OUT to en-queue packets to a specific switch port. The controller sends other control messages, such as OFPT_ROLE_REQUEST to change the controller's role in a multi-controller

environment.

OpenFlow Versions

The first version of OpenFlow (version 1.0) [24] was introduced in 2009 and supported single table and matched against 12 feilds. OpenFlow 1.1 [25] had full support for VLAN and MPLS tags and accommodated multiple tables. OpenFlow 1.1 also offered group abstraction with a group table. OpenFlow 1.2 [26] introduced flexible matching and provision to rewrite packet feilds with set_field extension. OpenFlow 1.2 also enhanced support for multiple controller and IPV6. OpenFlow 1.3 [27] released in April 2012, was the first long term release, supporting per flow meter tables and offered more flexible support for table miss flows. On-demand flow counters and duration for stats were other enhancements introduced in OpenFlow 1.3. OpenFlow 1.4 [28] supported optical ports, flow monitoring in presence of multiple controller and enabled role change notification for multi-controller setups. The latest version of OpenFlow 1.5 [29] released on 2015 supported packet type aware pipeline along with the provision to match against TCP bits.

This thesis uses OpenFlow 1.3.

2.2.1.3 Alternate SBIs

While the ability to program the data plane was the initial drive behind SDN and OpenFlow as SBI, other attempts were made before and after OpenFlow to make the data plane programmable. A brief discussion of these attempts follows.

Forwarding and Control Element Separation (ForCES) [30] advocates decoupling of the control element (CE) and forwarding element (FE) within a given network device but does not necessarily place the CE in an external logically centralised controller. CE controls and communicates with FE via Logical Function Block (LFB) provisioned in FE.

Ever-increasing protocol match fields in OpenFlow demand switches to understand

and parse an increasing number of protocol fields in packet headers. Authors of [31] argue that such a requirement is unnecessary and instead propose Protocol Oblivious Forwarding (POF) to make the FE a white box. POF is a protocol-agnostic southbound interface that addresses the high overhead in OpenFlow switches while parsing and matching OpenFlow header information. The authors of POF propose a generic, platform-independent flow instruction set (FIS). FIS allows the controller to parse the packets and only pass table-lookup instructions to the forwarding plane in search keys. FE extracts the search keys from the packet and uses information in FIS to perform actions.

DevoFlow [32] proposes to devolve control of most of the flows back to the forwarding devices while the controller retains control over a few significant flows. DevoFlow uses wildcards to reduce TCAM entries and the switch controller interaction instances. By pushing most of the control back to forwarding devices, FEs are enabled with local decision-making capabilities.

OpFlex [33] advocates pushing some of the complexities back to the data plane, in which the control plane computes policies centrally. The edge devices retain their intelligence as in traditional networks, while OpFlex formulates and pushes policies to the end devices.

[18], [34] provides a comprehensive survey of SBIs. A detailed discussion on programmable data plane can be found in [35].

2.2.2 Control Plane

The Data plane forwards packets based on flow rules in flow tables. With the data plane stripped of the ability of decision-making, flow rules must be populated in flow tables either manually or composed by an external entity. Manual configuration is unsustainable, poorly scalable, static and error-prone. Composition of flow rule by an external entity such as an SDN controller is automated, programmable. An SDN controller is a logically centralised entity that connects to, communicates,

and configures the SDN data plane. An OpenFlow SDN controller establishes and maintains a secure channel to each switch in the data plane using OpenFlow. The controller-switch communication is either in-band or out-of-band. A generic SDN controller architecture is presented in Figure 2.7.

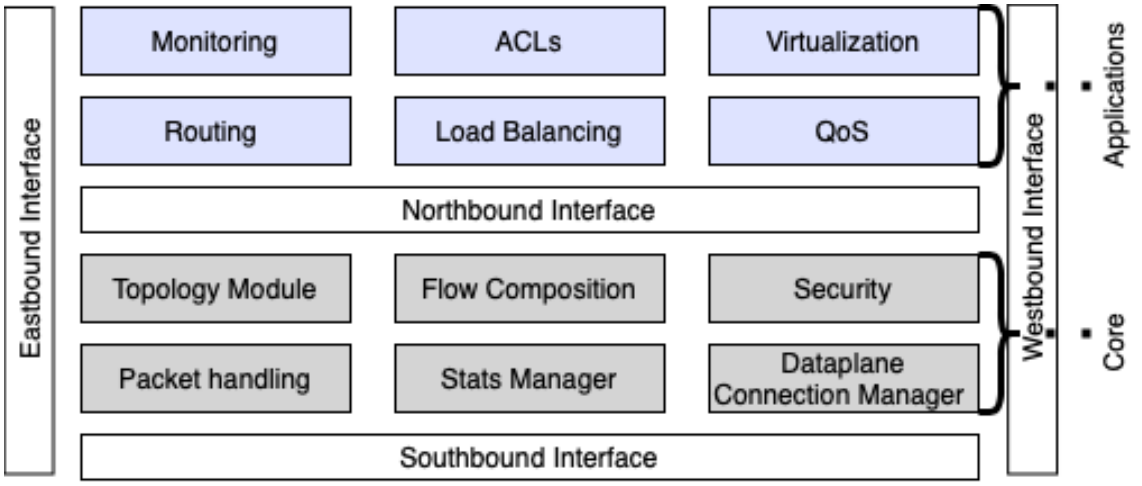


Figure 2.7: A generic SDN controller architecture

A controller's core components include I/O functions, event handling and essential components such as topology discovery and path-finding. The core component handles connections from switches, receives messages from switches and registers them as events for handling. The core also dispatches messages from various applications (NBAs) for further processing. Some applications include traffic policies, monitoring and generation of dashboards, load balancing and applications to maintain QoS. These applications use the information gathered by the core components to build their logic.

2.2.2.1 Control Plane architecture

From an architectural perspective, control planes are commonly classified into 1) Physically centralised architectures and 2) Physically distributed architectures. While centralised controllers are challenging to scale and prone to a single point of failure, consistency across multiple controller instances is challenging in distributed architectures. Below is a summary of a few well-known controllers in each category.

A detailed survey of architectures and performance of the control plane can be found in [18], [36], [37], [38].

Physically centralised architecture

A controller has all or most core components and applications on a single hardware entity. Such configuration immediately translates to scalability issues, single point of failures, zero-fault tolerance and bottlenecks. These issues have been documented in [39], [36], [40]. Below are a few well-known examples of physically centralised SDN controllers.

NOX [41] is the first open-source OpenFlow controller. NOX is a single-threaded, asynchronous, event-based controller. The controller core components include I/O operations, OpenFlow API, threading and event management components.

NOX-MT [42] is a performance enhancement on NOX [41] SDN controller. NOX-MT employs multi-threading and other optimisation techniques such as I/O batching, Boot Async I/O for handling incoming connections and fast processor-aware malloc to improve performance. Though NOX-MT only addresses some of the performance issues around NOX, it still outperforms the NOX controller.

Ryu [7] is also a component-based controller like NOX and NOX-MT. Like NOX, Ryu's core components consist of OpenFlow API to handle OpenFlow messages, event handler, memory management and messaging service.
Faucet [43] is a compact controller based on Ryu controller. Faucet has two primary components - Faucet controller and Gauge. While the controller connects to router module and other external systems, Gauge establishes a connection with the datapath, gather network statistics and feeds the information to visualisation tool Graphana [44].

Beacon [45] is a Java-based controller aiming at providing the run-time ability to start and stop applications running on the controller with high performance. To achieve high performance, Beacon implements a multi-threaded core.

FloodLight [46] is also a multi-threaded controller aiming at improving controller performance. Like the above-discussed controllers, Floodlight's core components include an I/O handler to read OpenFlow messages and generate events for handling by other controller modules. Other core components include a topology manager, a path computation, and REST API for accessing the modules.

While Ryu and NOX are single-threaded controllers, NOX-MT, Beacon, and Floodlight controllers are multi-threaded systems that take advantage of multiple cores [18].

Physically distributed architecture

Alternatively, a physically distributed SDN control plane contains multiple control entities handling the data plane. Based on how the control elements in the control plane interact, they can be further sub-classified as *logically centralised* and *logically distributed* [36].

Logically centralised SDN controllers maintain a consistent global network state. That is, each controller synchronises all local information with other controllers using a variety of state dissemination mechanisms to build a network-wide view amongst all controller instances. Below are some well-known logically centralised SDN control plane architectures.

ONOS [47] is distributed but logically centralised. One of the early primary challenges ONOS addressed was creating and maintaining a global network view. ONOS cluster consists of several ONOS servers. Each server instance discovers network topology, collects host information, and constructs a global view. The authors of [47] mention that the prototype of ONOS adopted modules from FloodLight [46]. While the network view is implemented using the Titan graph database, the Cassandra Key-value store is used for the distribution and persistence of information.

ONIX [48] states that "the principal contribution of Onix is defining a useful and general API for network control that allows for the development of scalable

applications". In this process, the authors propose an Onix API that provides functions for programmers to use, a control logic that determines network behaviour and runs on top of Onix API, and Onix distributed system that runs on a cluster of one or more physical servers. Onix stores network states as Network Information Base (NIB), which are read and written on by ONIX applications.

Hyperflow [49] is an application on NOX [41]. Hyperflow is designed as a distributed event-based, logically centralised and physically distributed control plane. The Hyperflow application and the event propagation system establish inter-controller communication. When a local controller receives a state-altering event, this event is published to construct a global view, and the rest of the controllers use this information. In case of a controller failure, the switches connect to the nearest backup controller.

KANDOO [50] control plane architecture is hierarchical. There are two levels of controllers. a) local controller b) root controller. Local controllers run local applications and consult root controllers only when they need to run a global application. The authors mention that KANDOO controllers (local controllers) cannot run applications that need global information in isolation and consult the root controller. The root controller runs instance of HyperFlow or ONOS controller. The local controllers do not appear to communicate amongst themselves and rely on the root controller for global information. Hence, KANDOO is logically centralised and not distributed.

OpenDayLight (ODL) [51] was initially conceived to provide a control platform to manage both OpenFlow and non-OpenFlow devices. ODL is a generic and general-purpose controller administered by the Linux Foundation. ODL clustering is a later release of ODL aimed at distributed deployment. AMQP is used for inter-instance messaging.

Elasticon [52] is logically centralised control plane consisting of a cluster of controller servers where each server is responsible for the data plane. All controller servers form a mesh using TCP connections, and a centralised module migrates

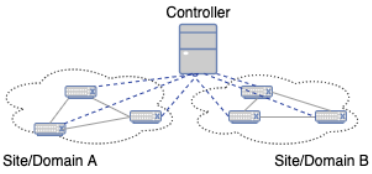
switches between servers. Information amongst servers is stored through Hazlecast distributed data store [53].

Logically distributed SDN controllers consist of control elements controlling some parts of the network (as in domains) and communicating on a need basis. Examples of such architectures are as follows:

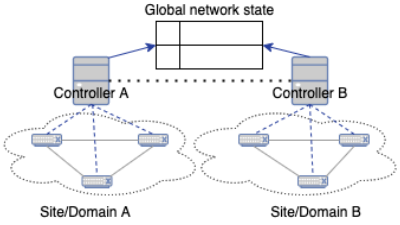
DISCO [54] views data planes as domains where a single controller controls one domain. DISCO has two main components. Intra-domain controller and inter-domain communication mechanism. The intra-domain controller is built on FloodLight [46] controller, and the messaging module is an application running on the floodlight controller which exchanges relevant information with other controllers.

D-SDN [55] physically and logically distributes control using a hierarchy of controllers. There are two sets of controllers - primary and secondary controllers. The main controllers delegate their actions to the secondary controller. Only those secondary controllers authorised by the primary controller can function on behalf of the primary controller. Secondary controllers can also act as fail-over controllers for other secondary controllers.

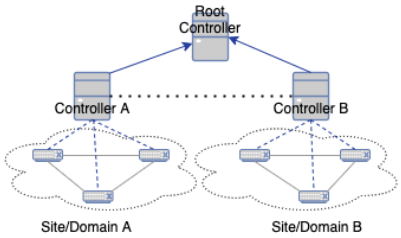
ORION [56] is a hybrid hierarchical control plane aiming to reduce computational complexity. There are two control planes, a lower-level area controller handling requests from the data plane and a higher-level domain controller handling information from area controllers. Area controllers collect device information, link information, process intra-area requests, and build network views. More importantly, the area controller abstracts these views and passes them to the domain controller. A domain controller can handle multiple area controllers and sees area controllers as nodes, stack the abstract views passed on by area controllers, and compute abstract paths. Area controllers cannot directly communicate with each other and can only communicate via a domain controller, while domain controllers use a distribution protocol for information dissipation.



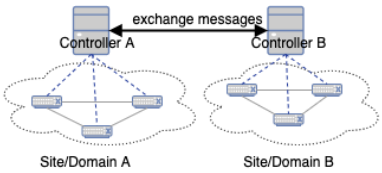
(a) Physically centralised SDN control plane



(b) Physically distributed, logically centralised SDN control plane



(c) Physically distributed, hierarchical SDN control plane



(d) Physically distributed, logically distributed SDN control plane

Figure 2.8: SDN control plane architecture

2.2.2.2 Northbound Interface

Northbound Interface (NBI) establishes a communication channel between the control and application planes. While there is no consensus on the standardisation of NBI, unlike SBI, some NBIs provides a high-level abstraction for application development [34]. Some of the well-known NBIs are discussed below:

Frenetic [57] is a language for programming OpenFlow networks. Frenetic has two sub-languages. First, *network query language* is used explicitly to install flow rules for querying the network state. Querying packets using high-level patterns result in the generation of an event. Such events are used in a variety of applications. Queries can also be composed to avoid overlap of flow rules and suppress superfluous packets to avoid race conditions. Second, a *network policy management library* generates policies to install flow rules on a switch.

Pyretic [58] allows programmers to specify network policies using high-level abstractions, allowing programmers to specify rules at the network level rather than for individual switches. Pyretic uses *predicate policies* to distinguish packets based on their locations, modify packet headers and compose flow rules in a serial and

parallel fashion. Pyretic facilities write dynamic policies and complex applications while inheriting the modular nature of Frenetic language.

Procera [59] is a controller architecture that allows network programmers to express high-level policies. Procera is reactive in a sense reacting to signals. Signals are transformed by signal functions defined by programmers and generate incoming network events. Core language is based on *functional reactive programming*.

NetCore [60] is a high-level declarative language to express packet-forwarding policies in SDN. Similar to pyretic, it uses predicate logic to forward packets to a set of locations. Complex predicates are built using logical operators such as union and intersection. NetCore provides compilation algorithms and a run-time system to install flow rules.

Flowlog [61] is a tireless language to support controller programming. A Flowlog program compilation consists of prepossessing rules to check compatibility and identify weaknesses, followed by an evaluation to extract predicates for composing a netcore policy.

NetKAT [62] is based on Kleen algebra to build primitives forwarding, modifying and filtering packets.

SCOR [63] proposes constraint-based NBI to address complex routing problems. SCOR defines predicates such as network path, link capacity constraint, residual capacity, path capacity constraint, path cost, delay, congestion, and link utilisation. By solving a combination of predicates, various network applications can be realised. For instance, a feasible flow path can be identified by combining and solving network path and link capacity constraints.

Apart from the above network programming languages, few controllers provide custom NBIs. PANE [64] adopts an idea of *participatory networks* where the network provides users or *principles* a configurable API. Application developers, end users, and applications are all considered principles. While all principals can create policies, policy conflicts are resolved using Hierarchical Flow tables. Thus PANE aims to offer

flexible mechanisms while resolving policy conflicts. ONIX is another such controller that provides a general API.

ONOS uses standard APIs such as Representational State Transfer (REST)API and JAVA for programmer interaction with applications. ONOS also uses Remote Procedural calls (gRPC) to enable distributed applications.

2.2.3 Applications

Authors of [18] categorise network applications into five categories: traffic engineering, mobility and wireless, measurement and monitoring, data centre networking and security and dependability. Some applications in traffic engineering and network monitoring, measurement and management category are as follows.

Traffic engineering applications perform forward / route packets. These applications forward packets on certain criteria, for example, to maintain QoS, schedule flows, load balance, or route packets on least cost paths. Hedera [65] propose a dynamic flow scheduler for multi-stage switch topology to maximise aggregate network utilisation. It does so by looping to find large flows at edge switches, estimating the natural demand of large TCP flows and using placement algorithms to compute good paths. Authors of [66] proposed Dynamic Flowentry-Saving Multipath (DFSM) to satisfy latency demands between data centres (DC) in a Wide Area Network (WAN). By releasing resources from over-fulfilled DC pairs and allocating them to each unsatisfied DC pair, latency demands are satisfied. Work presented in [67] proposes Weighted Cost MultiPath (WCMP) routing, where flows are distributed within a port group based on port weights. The weights assigned to each port are proportional to the capacity of paths associated with that port. Work presented in [68] propose Automatic Rerouting with Loss Detection (ARLD) to flag congested links, build virtual topology only with valid links, find alternate paths in the new topology, and assign flows in the new path. Authors of [69] proposes SD-FAST. Unlike previous solutions, SD-FAST is implemented at the switch. Bidirectional Forwarding Detection

is used to identify ports and SD-FAST updates flow rules with backup flow rules from OVSDB. [70] addresses congestion in the data centre by rerouting large flows. Flow-bender [71] is an end-host-driven load balancing scheme where congested flows are rerouted. Each TCP socket keeps track of a value V that is inserted into a flexible hashing field, the per-RTT fraction of marked ACKs as F. By using the value of F, a potential congested flow is detected. [72] propose a flow-scheduling algorithm. The algorithm aims to minimise network cost, which is the sum of convex functions of link utilisation. The algorithm performs better than ECMP. Mahout [73] proposes a Mahout controller and data plane shim. The shim in the data plane detects elephant flows, marks them and forwards them to the Mahout controller for appropriate action.

Monitoring and measuring applications gather network statistics regarding throughput, packet loss, delay and jitters. This information is made available for other applications. Network statistics are measured either actively or passively. During active measurements, additional traffic is injected into the network, and the behaviour of these packets is observed. On the other hand, passively measuring network traffic involves observing and not injecting any additional packets. Some of the monitoring and measuring applications designed for SDN are discussed here. Most of the applications make use of provisions in OpenFlow to monitor the network. OpenNetMon [74] measures per-flow metrics by querying the first and last switches in a path to retrieve flow statistics for predefined link destination pairs. Similarly, packet loss is estimated by polling the switch's ports. Authors of [75] propose a network-state management service-Statesman. Stateman allows maintaining three different network states- Observed state, target state and proposed state. While Applications use the observed state, the target state is desired network state that Stateman is responsible for updating. The proposed state is an intermediate state to allow conflict-less transition between the observed state and the target state. [76] propose a technique to measure available bandwidth using port stats captured by OpenFlow.

Realising the need for SDN control plane disintegration, a few researchers suggested alternate controller architectures. [3] suggested dis-aggregating SDN controller and extending the idea of externalisation of packet processing to external event processing mechanisms presented in [77]. The system uses Apache Kafka [78] to receive a stream of incoming packets and distribute them on the Kafka cluster for consumption by external applications. An application subscribes to listen to a specific packet event; upon the occurrence of this event, Kafka message distribution messages pass the packet to a possibly externally hosted application via the Kafka cluster. The application uses gRPC or REST API to install flow rules.

[79] proposes LEGOSDN, a fault-tolerant controller framework. Each application and controller core runs in their processes, and a component - AppVisor handles the exchange of SDN events and control messages between the controller and applications. This solution predominately detects application crashes wherein all network changes are rolled back and the application is restored.

[80] proposes μABNO, a cloud-native architecture for optical SDN controllers. The controller is decomposed into microservices and interacts with gRPC protocol based on ONF [22] transport API. Kubernets [81] manages the microservices. Some of the microservices are a) a connection microservice to compute paths, b) a virtual network topology Manager (VNTM) microservice c) a connection microservice, which then configures the necessary network configuration element. Thanks to Kubernetes managing the microservices, auto-scaling and auto-healing are inherent characteristics for the SDN controller.

[5] also propose a microservices-based SDN control plane. The controller and applications are built and run inside docker containers that communicate over messaging channels; furthermore, the controller core comprises multiple yang modules.

[82] propose Zero-SDN, flexible and modular architecture to achieve full-range distribution of event-based network control. The architecture is based on *micro-kernel architectures* where the controller logic is split into lightweight network modules-controllets. Controllets run in individual modules, are possibly distributed and can

communicate amongst themselves. This architecture stands apart in a way that controllets are also running locally on switches and all controllets communicate amongst themselves using a unified message bus - ZeroMQ [83].

[84] also propose a micro-services-based SDN controller architecture where internal controller components are decomposed into microservices. Micro-services use REST API for communication while gRPC and Web-socket are other alternative communication interfaces. The system is based on the Ryu SDN controller, where Ryu applications are containerised. [85] proposed a multi-agent-based autonomic network management architecture, where functions at each layer (Infrastructure, control and virtualisation and application layer) are recast into agents.

2.3 Conclusion

This chapter presented a brief overview of Software Defined Networks. Control planes are either logically centralised or distributed and operate on a single physical node or multiple nodes. Most existing controllers are monolithic in design, with tight coupling between the controller core and components. Multiple applications running on the controller are composed in sequence or parallel. Nevertheless uses OpenFlow's primitives to communicate with the data plane directly. More recently, modular control plane architectures have been proposed to disintegrate control planes using technologies such as microservices. The next chapter introduces the reader to multi-agent systems, following which the thesis advances to building a modular control plane as an agent system.

Chapter 3

Multi-Agent Systems

3.1 Introduction

Distributed System as defined by Steen in [86] *is a collection of autonomous computing elements that appears to its users as a single coherent system.* They are characterised by the following:

1. Concurrency: ability to run multiple processes simultaneously.
2. Isolated failures: Failure of one module in a distributed system does not affect the operation or functioning of other modules in the system
3. Scalable: Possible increase in system capability to meet a computing element's demand.
4. Social: Modules exchange information amongst themselves by sending messages.

Computing elements in distributed systems are software components. [87] defines a software component as an independent software unit that can be composed of other units to create a software system. Components have interfaces that define what services the components expect from other components and what services they can offer to other components. The internal methods are invoked by external components

via an interface, and all interactions must happen through these well-defined interfaces. A component's methods can be accessed or invoked remotely. A middleware intercepts remote calls and passes them to appropriate components. Middlewares, such as COBRA [88], are implemented as a set of libraries installed on each host system, along with a run-time system to translate calls and manage communications between components. Middleware in distributed systems supports two major interaction patterns.

1. Procedural-based: In Remote Procedure Calls (RPCs), components call remote procedures on other components using globally unique component names.

2. Message-based: In message-based interactions, a component creates a request message for another component. The receiving component responds with relevant information if any.

3.2 Multi-Agent Systems

Multi-agent systems (MAS) have been around since the late 1980s. They are distributed systems [89] with no centralised control. MAS comprise agents, and agents communicate with each other. Agents are concurrent, loosely coupled, distributed and isolated. They are either physically distributed or centralised, allowing flexible resource allocation (processing and memory) and large-scale computing. Agents are isolated; thus, any agent can go offline without affecting the entire multi-agent system allowing graceful degradation and repairs. The autonomy of decision-making also characterises agents, ability to reason and learn and react to environment perceptions [90].

What is an agent

There is no standard definition of an agent. [91] defines an agent as an entity in an environment that *perceives* the surroundings and acts to change the state of the

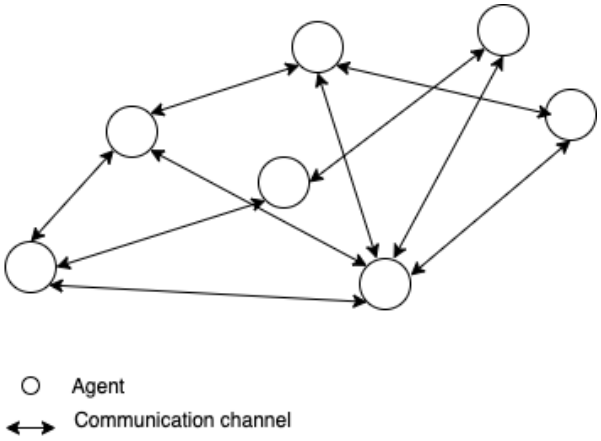


Figure 3.1: Multi-agent system

environment. [92] defines an agent as " a computer system that is *situated* in some *environment* and that is capable of *autonomous action* in this environment in order to meet its delegated objectives". [93] classifies and propose a minimum list of three attributes agents possess -autonomy, learning and communication using an agent communication language. Based on the attributes that agents possess [92] pens terms *weak agency* and *strong agency*. Generally, any hardware or software-based computer with autonomy, social ability, reactivity and pro-activeness has a *weak agency*. *Strong agency* also includes 'human-like' mental properties such as possessing knowledge, beliefs, intentions and rational choices to the properties mentioned earlier.

Any entity that can respond to environmental changes is an autonomous agent if the agent performs such actions without the influence of external entities (humans, agents, environment). When such an agent makes choices while working towards a goal and always makes the right decisions, then it is a rational agent. Learning agents explore unknown environments and learn from percepts.

An agent can fall under any of the agent types mentioned; all the agents generally possess the ability to socialise with other agents using agent communication language and behave autonomously to a certain extent.

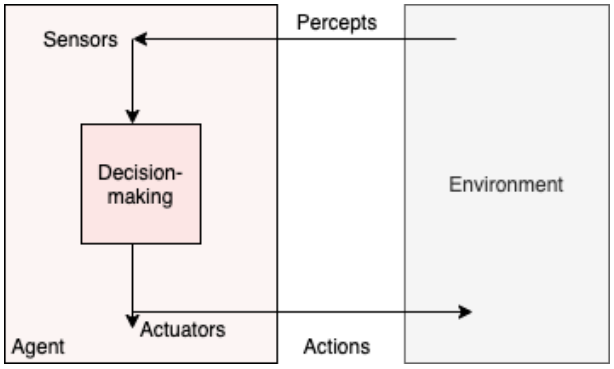


Figure 3.2: Agent's interaction with Environment [91]

3.2.1 Software Agent Architecture

An agent comprises a reasoning module, a knowledge module, a communication module, and an optional learning module. [94] broadly classifies software agents as deliberative or reactive. [91] proposes a third and advanced type - a learning agent.

3.2.1.1 Reactive agents

1. *Reflex* agent selects actions based only on current percept. Actions are mapped to precepts as rules, and the agent performs a specific action mapped against a percept.

2. *Model based* agent is a reflex agent with an internal stored percept history state. Thus, the agent takes decisions based on stored history and current percept.

3.2.1.2 Deliberative agents

1. *Goal-based* agents set a target goal and perform a sequence of actions to achieve this target state. Unlike reflex agents, goal-based agents contemplate based on current percepts and act. Though it might take longer, this contemplation is more flexible than reflex agents as no set rules are defined.

2. *Utility based* agents quantify how well a goal has been achieved using some performance measure. These agents try to maximise the utility function while achieving a goal.

3.2.1.3 Learning agents

Learning agents have an advantage over reactive and deliberate agents as they can operate in unknown environments and learn from this experience—both reactive and deliberate agents can learn using an additional learning component.

As mentioned earlier, these are only some of the available definitions for agent architecture. [92] classifies agents as reactive, deductive, and practical. While reactive agents are simple reflex agents, deductive agents use logical languages such as PROLOG [95] to represent and reason about the environment. On the other hand, practical agents employ the Belief-Desire-Intentions (BDI) methodology to reason about actions to perform. The agents are called as practical agents, since they weigh actions available and decision is influenced by the beliefs and desires the agents hold. In general, the agent's knowledge about the environment is *Belief* of the agent, while *Desire* is a goal state and *Intention* are intermediate steps to fulfilling a desire.

3.2.2 Reasoning

Agent architectures can be classified based on the environment the agents are operating in and the amount of knowledge available to the agent. While reflex agents do not necessarily reason, deliberate agents reason about their environment. [91] classify them as following.

1. *Problem-solving agents* search the solution space for a solution. These search methods include classical strategies such as uninformed and informed methods, sometimes using heuristics. Beyond such classical search strategies, these agents can also employ advanced techniques such as searching in unknown environments, searching in continuous spaces, employing techniques such as Alpha-Beta pruning and finally, employing Constraint satisfaction to find solutions [91].

2. *Logical agents*, on the other hand, represent the world in first-order facts and rules. Deductive reasoning and inductive reasoning are some examples.

3.2.3 Learning

Machine learning techniques generally involve identifying actions that have arisen in response to a specific behaviour. Some of the learning techniques employed by agents are:

1. *Inductive learning*: learning a general function or a rule from specific input-output pairs. Inductive learning programs combine logic and machine learning.

   (a) *Supervised learning*: available feedback enables learning of the function of form $y = f(x)$, where y is the target value for input, x. If the function is discrete, the method is called classification; if the function is continuous, it is called regression.

   (b) *Unsupervised learning*: No prior information on inputs or expected outputs are provided to the agent. The agent employs techniques such as clustering to learn the underlying relations between features and outputs. K-means clustering [96] is an example of unsupervised learning.

2. *Analytical or deductive learning*: a priori knowledge exists. Improving the knowledge base is based on data/feedback and extending or simplifying data. Also, from a known general fact to a new facts are logically entailed.

3. *Reinforcement learning*: enables the agent to try and learn from the feedback received from the environment. The feedback is either a reward or a punishment. A typical reinforcement learning setup is a *Markov decision process* having a state (S), transition probabilities (T), possible action set (A) and a reward function (R). Reinforcement learning is either model-based or model-free based on knowledge of the environment. The agent knows the transaction matrix (T) and reward function (R) in model-based learning. On the other hand, as in

most real-life environments, the agent uses model-free learning in the absence of a transition matrix and reward function. Q-learning [97] is an example of model-free learning.

3.2.4 Knowledge representation

Knowledge needs a formal representation to maintain consistent information and enable communication among agents. This formal definition of a body of knowledge is known as *ontology* [92]. Two main benefits of formalising knowledge are *reasoning* with a reasoner to deduce and to *standardise* information representation across multiple agents and enable communication. Thus an ontology defines and represents concepts and establishes relations between these concepts. An ontology is expressed with varying degrees of expressiveness-ranging from a simple controlled vocabulary to a very complex formulation involving logical constraints. Most popular ontology languages include Knowledge Interchangable Format (KIF) [98], Extensible MarkUp Language (XML) [99] and Web Ontology Language (OWL) [100]. XML uses tags to define new vocabulary to identify entities. XML only provides the basic vocabulary to store information and does not perform advanced functions such as creating sub-classes and properties or defining relationships between different entities that need processing by XML applications. That is, sentences such as " A person who teaches is a teacher " cannot be represented using XML. KIF, on the other hand, is very expressive, but it is computationally complex [92]. While being human and machine understandable, OWL allows for automated reasoning to check ontology's consistency and identify any contradictions within the representation. This thesis represents knowledge in OWL language as discussed in Chapter 5.

3.2.5 Communication

Communication is an essential entity of any multi-agent system. Synchronising the agent's actions is necessary to maintain data and state consistency. [92]

distinguishes between agent communication and object communication. While object communication (software components) invokes other objects' public methods, agents, on the other hand, choose to act upon receiving a message. Thus, agents exhibit autonomous behaviour while communicating with other agents.

To achieve strong agency, communication in multi-agent systems uses performative verbs such as *request*, *inform*, *promise*, *believe* and *want*, among others. Multiple Agent Communication Languages (ACLs) have been proposed to enable inter-agent *conversations*. Predominant ACLs include Knowledge Query, and Manipulative Language (KQML) [101], which defines message formats for communication. KQML does not concern itself with the message's contents but rather just the format of the message and is a message-based language. FIPA-ACL [102] is another agent-communication language. FIPA, like KQML, defines performative. Performative conveys the intent of the message and the sender. For instance, a performative *inform* is used to inform another agent, and no response is expected. In an argument, performative such as *accept-proposal* is used.

3.3 Software agents vs software components

To summarise, components of distributed software are distributed across a network. A self-contained and isolated component must have necessary and well-defined interface. External components access public methods using the interfaces. A well-defined interface makes updating components seamless to external components as long as the interface does not change. A middleware enables and supports interaction (both procedural and message based) between the components. Interaction using procedural calls such as Representational State Transfer API (REST) [103] is synchronous. It requires both components to be online, whereas for message-based interaction, messages are stored in a message queue for the receiver component to use when available. Hence message-based interaction is asynchronous.

Micro-services, like software components, operate using well-defined APIs such as

REST/HTTP. Micro-services are incapable of proactively taking initiatives based on percepts from the environment. Instead, they react promptly on inbound requests from other agents.

Agents share some characteristics such as autonomy, composition and sociability with components. Nevertheless, an agent cannot invoke the methods of other agents. Unlike micro-services, agents interact with the environment to react and take actions proactively. Also, to achieve strong agency, agents use speech performative such as *request* and *negotiate*, which components are incapable of while maintaining a belief system.

Indeed, both microservices and agents are candidates for building a modular, disintegrated SDN control plane. Some earlier work in this area include [80], [5], [84], which propose a micro-service architecture and [82] present a micro-kernel-based SDN controller architecture.

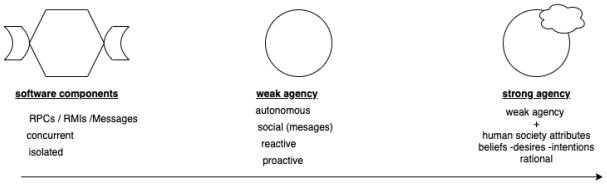


Figure 3.3: components vs agents

This thesis identifies computing entities as *agents* because they are autonomous, social and exhibit varying degree of agent intelligence. No external entity can influence how the entity operates, react to environmental changes (monitoring agents), and proactively try to achieve goals (management agents). In other words, the entities exhibit weak agency.

3.4 Conclusion

This chapter attempted to familiarise the reader with multi-agent systems and agents in general. The concept of an agent is discussed. Various agent architectures and agent abilities such as reasoning and learning are defined. Knowledge representation

and communication aspects are briefly discussed. Chapter 5 discusses knowledge representation and agent communication in detail.

Part II

Prototype

Chapter 4

MASDN Controller Architecture

4.1 Introduction

This chapter is the first of three chapters detailing the design and operation of the proposed MASDN controller. This chapter analyses and disaggregates the existing SDN control plane into agents by identifying agent roles, following which individual agent's behaviour is described in detail.

4.2 Breaking the monolith

The first step in breaking the monolithic SDN controller is identifying the responsibilities and goals of the controller. Next, sub-goals and tasks that entail fulfilling high-level responsibilities are identified. Each agent is assigned a role associated with performing one or more tasks. SDN controller's primary responsibilities are provisioning flows, monitoring the network and managing the network both reactively and proactively, as shown in Figure 4.1. The *goal* of the agent *ag* is to perform task t to fulfil a responsibility r. A *goal* is an abstract concept that captures *what* the agent needs to achieve but not *how*. Optionally, a performance metric (*utility*) measures how well an agent performs a task. Complex agents use the metric to improve their decisions.

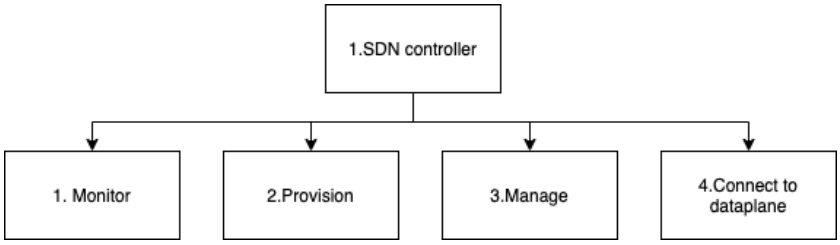


Figure 4.1: Primary responsibilities of SDN controller

Figure 4.2 lists high-level goals and sub-goals for monitoring activities. Mainly, monitoring responsibility has four high-level goals. They are 1.1 monitor port status, 1.2 monitor flow tables, 1.3 monitor port counters and 1.4 discovering topology. Goals 1.1 monitoring ports' operational status and 1.3 monitoring port counters are periodic, and an agent performs tasks periodically to fulfil periodic goals. All goals need not have precedence, but a sub-goal must be fulfilled to achieve a high-level goal. For example, only after the agent receives link information (sub-goal 1.4.1) and updates the knowledge base with the link information (sub-goal 1.4.2) do we consider goal 1.4 to discover the topology fulfilled. While most sub-goals identified are necessary goals of monitoring agents, sub-goal 1.3.1 is optional but demonstrates a complex agent in action.

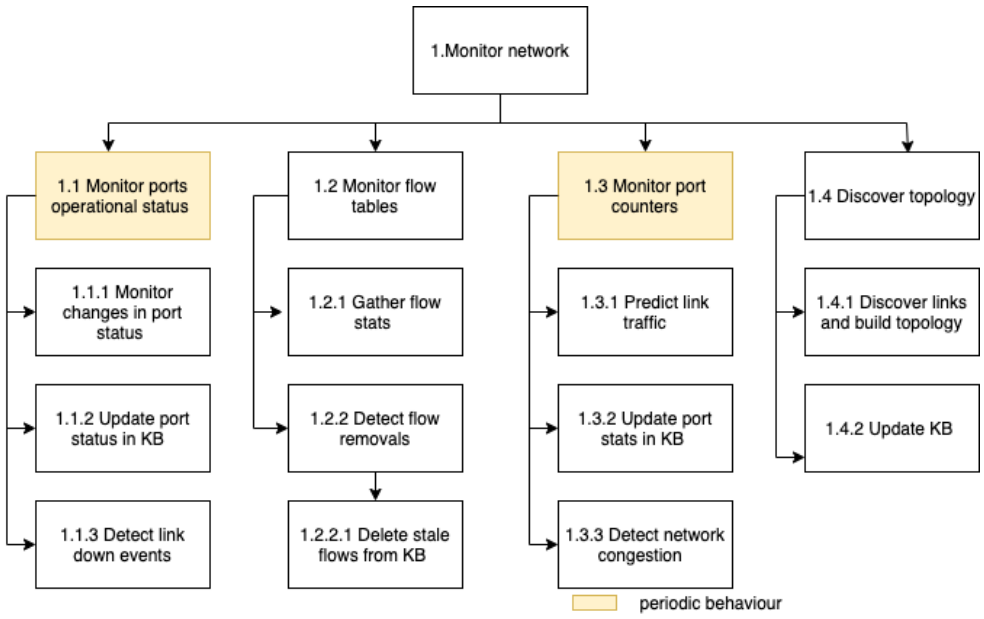


Figure 4.2: Goal hierarchy diagram for monitoring responsibility

Figure 4.3 lists SDN controller's goals and sub-goals of provisioning responsibility.

There are two primary goals 2.1 to provision a new flow and 2.2 to re-provision an active flow. Acquiring an updated topology and using a path-finding algorithm are two sub-goals of the primary goal 2.1, as shown in the goal hierarchy branch for 2.1 in Figure 4.3. The sequence of goals indicates precedence. A topology must be available (sub-goal 2.1.1) for the system to find the path between a source and a destination node (sub-goal 2.1.2) using any path-finding algorithm. Goal 2.1.3 /2.2.1 is a common goal.

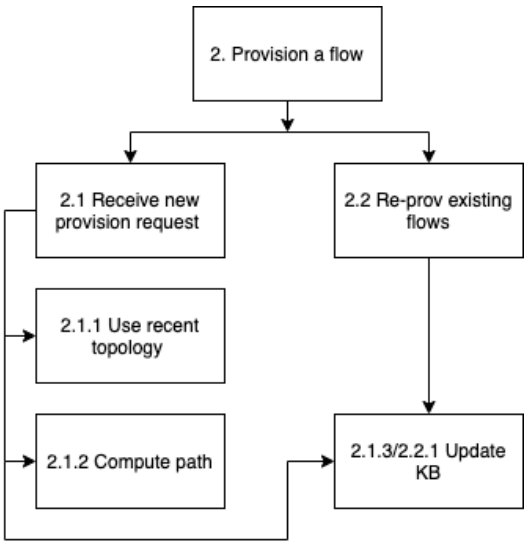


Figure 4.3: Goal hierarchy diagram for provisioning responsibility

High-level network management goals are 3.1 recovering from link failures, 3.2 alleviating congestion and 3.3 ensuring fairness amongst TCP flows. While goal 3.1 is necessary for network management, goals 3.2 and 3.3 are optional and demonstrate complex agent integration and operation in the agent system. The system must first identify network abnormalities (captured in sub-goals 3.1.1, 3.2.1 and 3.3.1) as a prerequisite for managing network abnormalities. Flows may be rerouted to alternate paths while not shifting congestion to other network parts to alleviate congestion on a few links. In contrast, in the event of link failures, rerouting flows on available paths is prioritised over adhering to bandwidth constraints. Goal 3.3 ensures TCP fairness amongst heterogeneous TCP flows by adjusting sending rates. Figure 4.4 shows the hierarchy of the agent system's management goals.

Monitoring activities such as topology discovery, gathering port statistics, and

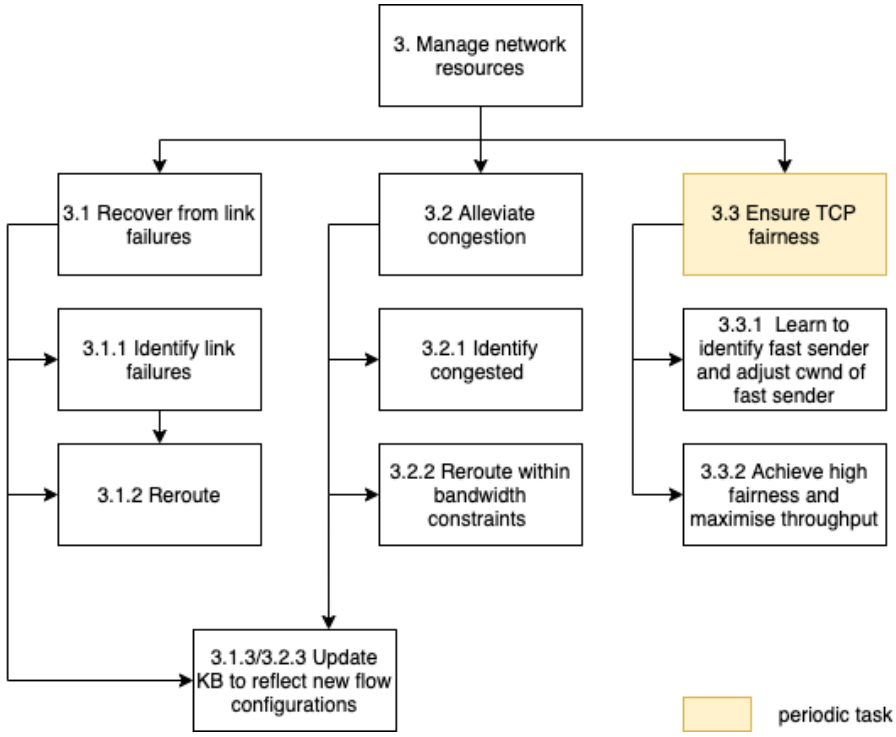


Figure 4.4: Goal hierarchy diagram for management responsibility

port status are periodic tasks and do not require complex logic. On the other hand, identifying congestion and link failures requires decision-making capabilities. Identifying unfair resource allocation and traffic prediction requires agents to possess learning capabilities. While this thesis builds use cases that require a varying degree of agent intelligence, the agent system is expandable to include additional agents to fulfil additional responsibilities.

The second step of MaSE methodology is applying use cases to demonstrate communication. Chapter 5 of the thesis describes the communication patterns and messages exchanged by the agents. Agent roles are defined in the third step. An agent's role fulfils a responsibility by achieving goals [15]. While there is usually one-to-one mapping between roles and goals, it is common for a role to achieve multiple goals. Commonly, a goal is distributed across multiple roles and requires communication. Figure 4.5 captures agent roles assigned based on the agent's responsibilities and goals agents achieve.

Different agents fulfil different roles. While micro-roles can be created for sub-goals (for example 1.4.1, 1.4.2), defined roles in the proposed agent system reduce excessive

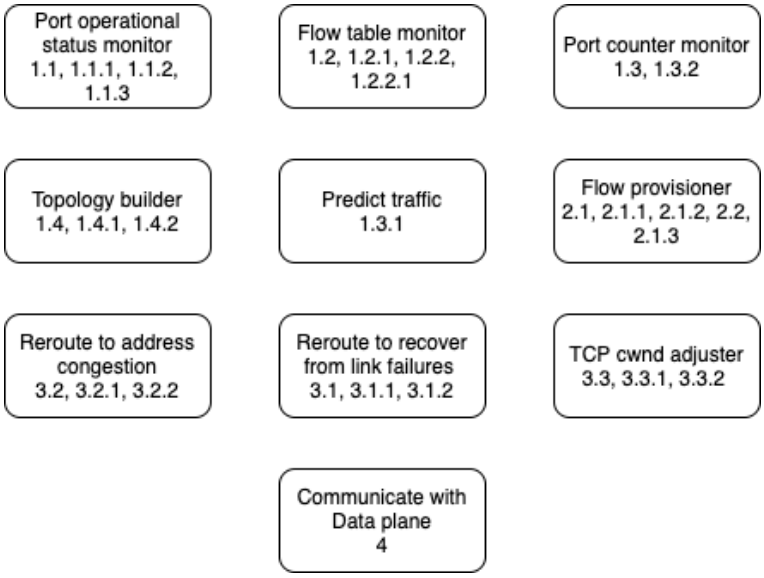


Figure 4.5: Agent roles

communication between agents and assign dependant tasks to the same agent. For example, goal 1.4.1 has two tasks - sending LLDP packets and processing the received packets. These tasks can be performed to two different agents. Such assignment not only increases agent communication but also introduces latency in the system. Instead, sending and receiving LLDP packets and updating the knowledge base with this information are all tasks assigned to a single role - topology builder.

The agents in the proposed MASDN control plane are broadly grouped based on their high-level responsibilities. Unlike traditional layered architectures, there is no top-down or bottom-up hierarchy in the layers of the proposed solution. Figure 4.6 shows a high-level architecture of the agent system. The connectivity between communicating agents is represented using colour arrows, and each colour indicates an end-to-end communication channel. Broadly,

1. South Bound agents interact with the data plane.

2. Provisioning agents provision new flows using packet information such as Virtual LAN (VLAN) ID, Source and destination MAC addresses or source and destination IP information.

3. Monitoring agents monitor various aspects of the network and maintain a

network state in the knowledge base.

4. Management agents manage the network and handle network anomalies by rerouting flows to balance traffic, adjusting congestion window, reroute affected flows in case of link failures.

5. Knowledge layer stores a global network state in OWL format.

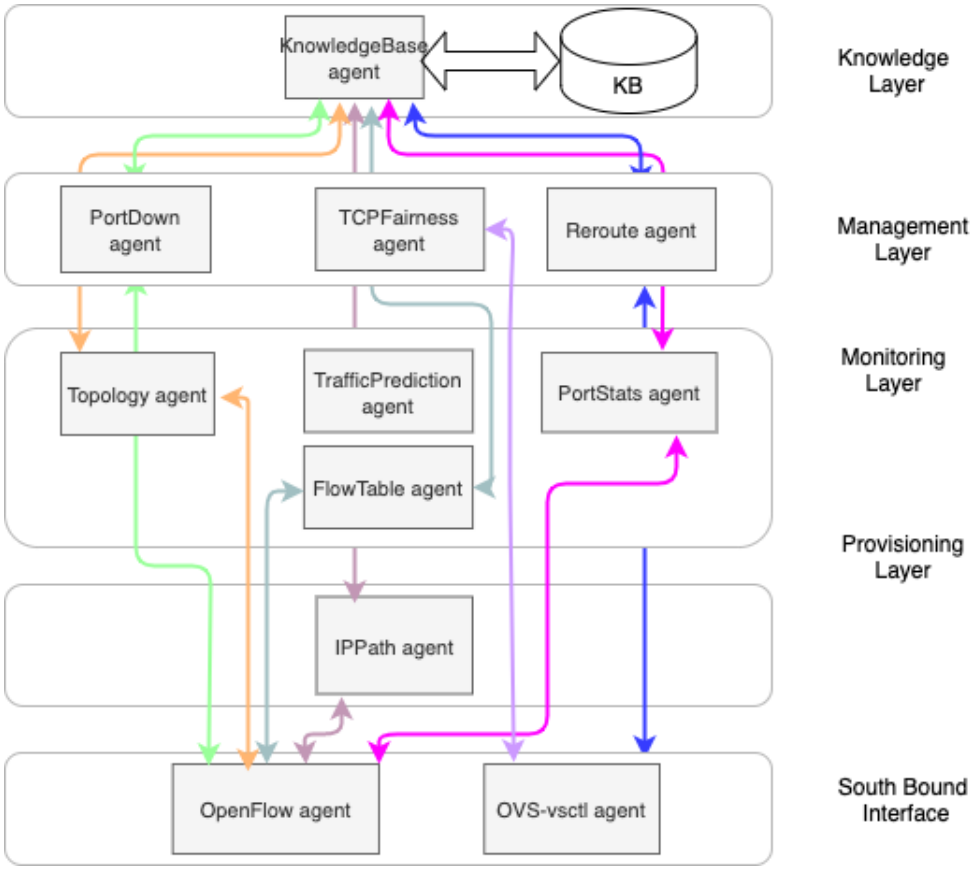


Figure 4.6: High-level architecture of proposed agent-based system

The upcoming sections discuss individual agent architectures. Some agents exhibit cyclic behaviour (those in monitoring and management layers) and others are reactive.

4.3 SBI Layer

This layer consists of agents communicating with the data plane and implementing the southbound interface. OpenFlow agent and ovs-ofctl agent are two agents operating in this layer.

4.3.1 OpenFlow agent

OpenFlow agent (OFagent) implements a TCP server to establish OpenFlow communication sessions with the data plane. OFagent maintains connections with the data plane, parses incoming OpenFlow packets and passes this information to other agents. OFagent also receives information from other agents, packs the information into OpenFlow messages and forwards it to the data plane. OFagent runs two event loops, each catering to the data plane and the agent system for exchanging messages.

OFagent receives a message, identifies the receiver of the message and forwards the message to the intended recipient. The agent is equipped to handle OpenFlow control messages such as OFPT_HELLO, OFPT_ECHO_REQUEST and OFPT_FEATURE_REPLY. For non-control messages such as OFPT_PACKET_IN, OFPT_PORT_STATS, OFPT_FLOW_STATS, OFPT_PORT_STATUS messages, the agent relies on other agents for processing. Such interaction with external agents to handle OpenFlow packets deviates from traditional SDN controller applications. Traditional controller applications consume OpenFlow packets and are thus tightly coupled with OpenFlow.

In contrast, the OFagent *outsources* packet handling to an external agent. OFagent's decision on where to forward the packet payload is based on the incoming packet's payload type. For instance, if OFPT_PACKET_IN's payload is a Link Local Discovery Protocol packet (LLDP), the agent forwards this information to the topology agent. Similarly, an IP packet received due to the absence of flow rules (and the presence of a table miss entry) is forwarded by OFagent to the IPPath agent

for provisioning the flow. Monitoring messages such as OFPT_PORT_STATS are parsed, and information is passed to PortStats agent. The agent forwards information obtained by parsing the incoming OFPT_PORT_STATUS packet to the PortStatus agent. The data plane sends unsolicited messages to inform the agent about a change in the network, for example, a port's status change or when flow rules are removed from the datapath due to timeouts. OFagent forwards this information to corresponding agents responsible for updating the global knowledge base. In case of a port status change from *up* to *down*, the OFagent forwards the information to the PortDown agent, which handles the re-provisioning of affected flows. The agent creates OpenFlow messages as a response to requests from other agents. For example, the agent creates and sends OFPT_PORT_STATS messages to the data plane upon receiving a request from the PortStats agent. Similarly, the agent creates OFPT_FLOW_MOD messages to add and delete flow rules from the data plane. OFagent's operation is summarised in Figure 4.7.

Monitoring and management agents do not necessarily have to interact with the data plane using OFagent. Indeed, some agents in the management layer autonomously operate via the ovs-ofctl agent.

4.3.2 OVS-ofctl agent

OVS-ofctl agent (OVagent) provides access to datapath's flow tables using the OpenVSwitch management tool ovs-ofctl. Specifically, this agent accepts flow-stats requests from Reroute agent, formats the information and responds to Reroute agent. The agent also provided flow-table access to the TCPFairness agent to install flow rules. The agent's operation is shown in Figure 4.8.

While both OVagent and OFagent use OpenFlow protocol, OVagent agent access the switch's flow tables using the ovs-ofctl tool and not OpenFlow messages. OVagent and OFagent in the SBI layer demonstrate the idea of concurrent access to the data plane by multiple SBI agents. NETCONF [8], YANG [104] agents are other possible

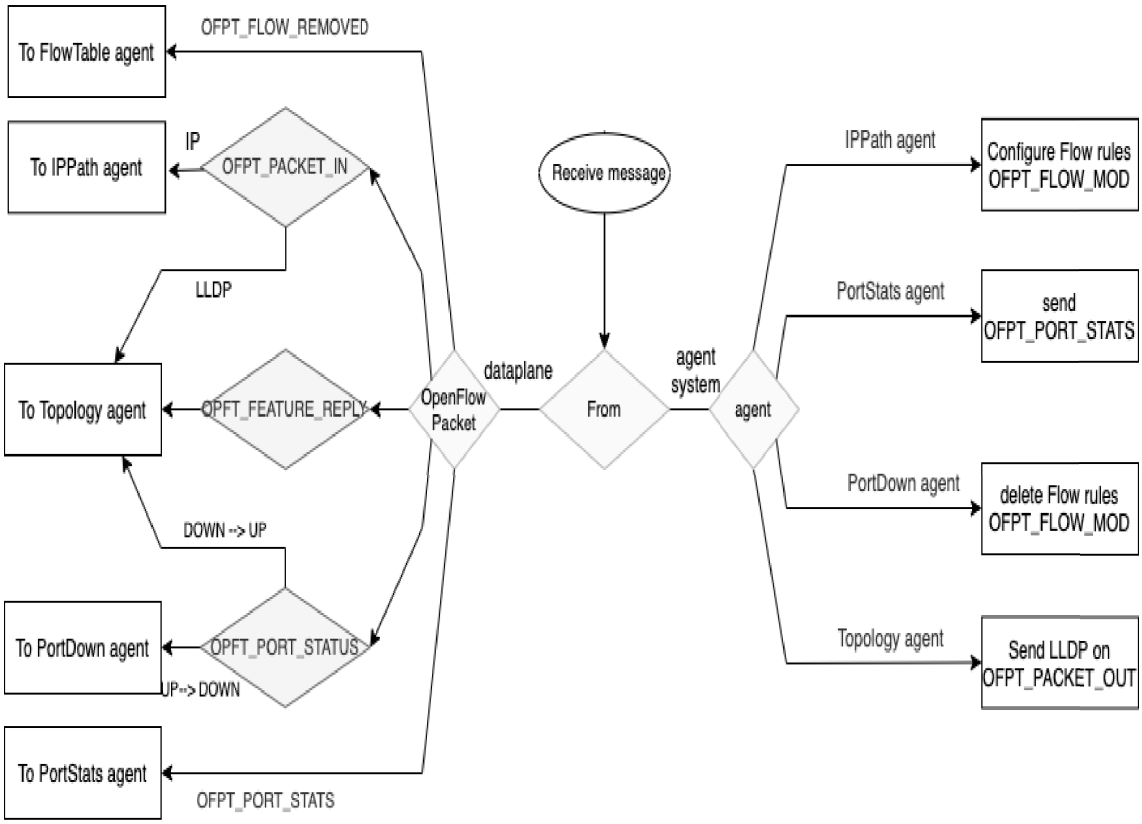


Figure 4.7: OpenFlow agent

additions to the agent system aimed at managing traditional (non-OpenFlow) devices.

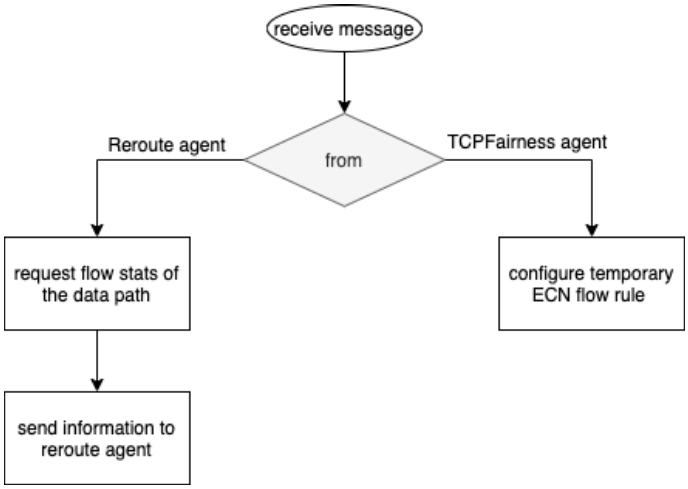


Figure 4.8: OVS-ofctl agent

4.4 Provisioning Layer

4.4.1 IP Path agent

As the name suggests, the IPPath (IPagent) agent primarily finds paths to provision flows. A*flow* is a stream of packets originating from a source node towards a destination node. The agent computes complete end-to-end paths and provides this information to OFagent for flow installation. Path information of form $\{(dp_1, pr_1), (dp_2, pr_2), ..., (dp_n, pr_n)\}$ is a *complete* path where dp_i is datapath ID and pr_i is outport of dp_i. IPagent springs into action after receiving provisioning and re-provisioning request from other agents, including OFagent, Reroute agent and knowledge base agents.

The IPagent agent implements Dijkstra's path-finding algorithm to provision a new flow. For re-provisioning requests from management agents, the agent computes complete path information and sends the new information to OFagent for re-configuration of affected flows. Finally, the agent maintains a local copy of the topology to compute paths. This local copy is obtained from the knowledge base agent and updated periodically. The knowledge base agent also pushes new copies upon any change in topology (such as port failures or the addition of new links). The operation of the IPagent agent is shown in Figure 4.9.

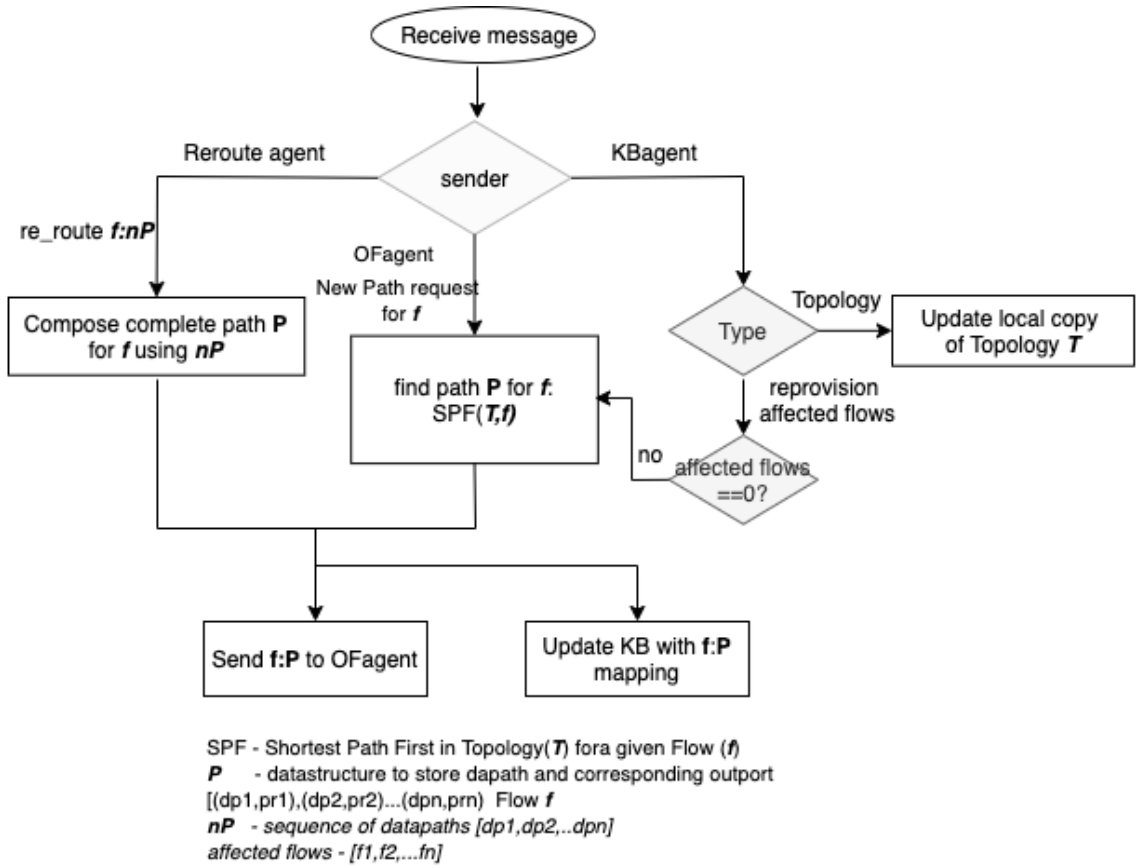


Figure 4.9: IP Path agent

4.5 Monitoring

Monitoring agents are a group of agents responsible for monitoring and maintaining an updated network view. The agents periodically query the network via OFagent and update the knowledge base with the information.

4.5.1 Topology agent

The topology agent (TOagent) periodically requests OFagent to rediscover topology. The TOagent processes the link information, composes the topology, and updates the knowledge base with the information. TOagent agent gathers a list of all datapaths and connected ports information from OpenFlow features response message. Using this information, the TOagent initiates the topology discovery process by requesting OFagent to pump LLDP packets into the data plane. The received LLDP responses

are processed to compose links between datapaths. A network graph is created using the link information. Link information and graph are stored in the knowledge base.

4.5.2 PortStats agent and FlowTable agent

PortStats agent (PSagent) estimates the current throughput and available bandwidth on all links. As mentioned in chapter 2, network monitoring is either active or passive. Active probing techniques involve sending a probe packet regularly to gauge the network state. Works of [105], [106] estimate available bandwidth using active methods in traditional networks. Authors of [107], [76], [108], [109], [110] estimated traffic or available resources in SDN using active methods. On the other hand, passive techniques use statistical models to estimate available bandwidth as listed in [111]. SDN allows active monitoring of OpenFlow switches, and a study conducted in [112] suggest that pure SDN-based results are sufficient for selected applications.

PSagent periodically requests OFagent to collect port counter information (transmitted bytes tx_n and received bytes rx_n) from all ports. The agent queries for port counters every t secs (for example 15 secs), and collected port stats are stored in the global knowledge base. The bandwidth capacity of the link bw_l is the maximum number of bits that the link l can transfer for a unit of time and is usually measured in bits per sec (bps).

Thus, for a link l with bandwidth capacity bw_l, link throughput th_l is the actual number of bits traversing the link. Link throughput is computed using two consecutive port counter readings x_n, x_{n-1} bits at times t_n and t_{n-1} instances and given by Formula 4.1.

$$th_l = (x_n - x_{n-1})/(t_n - t_{n-1}) \tag{4.1}$$

Link utilisation u_l is the percentile of bandwidth capacity used and given by

Formula 4.2.

$$u_l = (th_l/bw_l) * 100 \tag{4.2}$$

Finally, the available bandwidth of a link abw_l is leftover or unused bandwidth given by Formula 4.3.

$$abw_l = bw_l - th_l \tag{4.3}$$

IPagent sets an idle timeout on all the flow rules and enables flow-removal notifications for all the flow rules. Thus, when a flow is no longer active, FlowTable agent FTagent receives flow-removal messages from OFagent and purges the knowledge base. This ensures updated flow information in the knowledge base.

A generic monitoring agent's operation is shown in Figure 4.10.

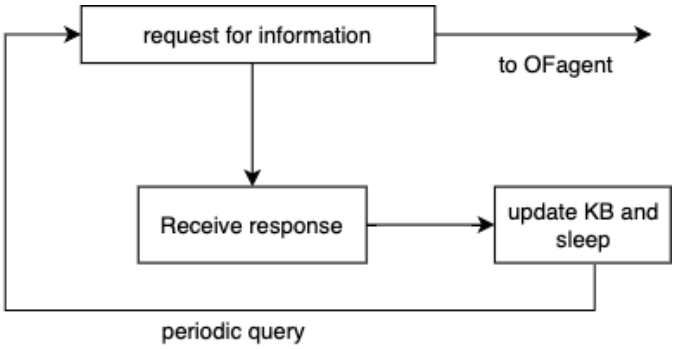


Figure 4.10: Generic monitoring agent

4.5.3 Traffic prediction agent

Predicting network traffic is a desirable feature for better network management. While long-term predictions over days or months allow better network resource planning, short-term predictions over minutes and seconds have an immediate effect on the network caused by proactive network management. In the past, linear and non-linear prediction models were applied to predict network traffic patterns. Linear

techniques consider network traffic as a time series and apply forecasting models such as Auto-Regression (AR), Moving Average (MA), Auto-Regression Moving-Average (ARMA) and Auto-Regression Integrated Moving-Average (ARIMA). Non-linear methods apply machine learning techniques - both classical and Deep Learning techniques to predict network traffic.

4.5.3.1 Related Work

Authors of [113] apply AR, MA and ARMA to predict network traffic. Auto Correlation Function (ACF) and Partial Auto Correlation Function (PACF) plots are used to ensure the stationality of the series. The authors conclude that AR makes the best predictions for daily traffic.

[114] apply ARIMA on observations made with SNMP made every 30 secs and work presented in [115] perform video traffic prediction using Fractional Autoregressive Integrated Moving Average (FARIMA). The proposed method considers the self-similar nature of network traffic, which exhibits long-range dependence.

In [116], the authors propose a combination of linear techniques such as ARIMA with non-linear Generalised auto-regressive conditional heteroscedasticity (GARCH) model to accurately predict traffic while capturing long-range dependence characteristics of network traffic.

Work presented in [117] employs non-linear methods such as Long Short Term Memory (LSTM) to predict network traffic. Though the technique marginally decreases the packet loss ratio, a higher delay is observed due to resource-intense training of the neural network.

[118] propose Deep Neural Network based traffic prediction with bandwidth data aggregated over 60 minutes and predicting every hour.

Authors of [119] propose to apply Bidirectional LSTM and Bidirectional Gated Recurrent Unit (GRU) and use Mean Squared Error (MSE), Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE) as metrics to validate the model. The

solution also employs Convolutional Neural Network (CNN) for feature extraction. While pure GRU and Bidirectional-LSTM did not outperform pure LSTM-based predictions, their hybrid proposal results were comparable to pure LSTM-based proposals.

Most prediction models use an available dataset created over time to make predictions. Such mechanisms are untimely and require extensive training, specifically in non-linear techniques such as the application of neural networks. The dataset is usually divided into training and test set, and data characteristics such as stationality and correlation are understood and dealt with. A model thus obtained is then used to make predictions. This type of learning is known as offline learning or batch training.

4.5.3.2 Incremental learning

Usually, a machine learning (ML) model trains on the training dataset. Training generally involves adjusting the model parameters based on a cost function. In *batch training*, the model stores the entire dataset in memory and model parameters are adjusted over the entire dataset. Once a model is trained, the model is retrained with the next batch of some fixed window of size w. Batch training suffers from high latency caused by processing large data sets resulting in untimely predictions for most recent data [120]. Batch learning is also called *offline learning* as the model is trained only in intervals when a new batch of data is available.

Now, consider the problem of predicting traffic on a link. The PSagent agent reads port stats periodically, creating a continuous *stream* of data. Consider a j^{th} entry in a batch of size w collected at time t_i. The next predicted value is for time instance $t_{i+(w-j)+1}$ and not at t_{i+1}, which is untimely, as the model trains over the entire batch and waits until all data points are available.

An alternative to offline learning is *online learning* (incremental learning), where the model does not require a dataset in batches of size w and is trained for every

incoming data point in real-time. In other words, the w is 1. Online learning is designed for machine learning models to learn from a continuous data stream. Unlike offline learning, an online learning module starts learning with just one data point. A single data point is loaded in memory and processed at any given time, requiring fewer resources than an offline process. Real-time prediction, allowing data evolution at any time and catering to infinite data streams without resource restrictions are some known benefits of incremental learning [120]. These benefits make incremental learning a suitable learning mechanism for traffic-predicting agents.

In **linear regression** [121], an output (y) is a linear function of inputs (x). A linear regression model is of form Equation 4.4:

$$y_t = w_0 + w_1 x_{1,t} + w_2 x_{2,t} ... + w_k x_{k,t} + b_0 \tag{4.4}$$

where y_t is output, $x_{1,t}$,$x_{2,t}$,...,$x_{k,t}$ are inputs at time t and w_0,w_1,w_2,...,w_k are parameters.

Most standard techniques, such as ordinary least squares, computes the distance between the actual value and the regression line generated by the model to estimate the weights w_i. These parameters are adjusted to reduce the distance between actual and predicted values. This algorithm is known as Gradient Descent and requires the availability of an entire dataset.

Linear regression parameters w_0,w_1,w_2,...,w_k can also be adjusted incrementally for each iteration using Stochastic Gradient Descent (SGD) [122]. Unlike gradient descent that batch learning uses where gradient over the entire dataset is computed, SGD computes gradient over a single data point to reduce cost function given by Equation 4.5 :

$$w_{t+1} = w_t + \alpha_t * \bigtriangledown(Q(x_t, w_t)) \tag{4.5}$$

where w_{t+1} and w_t are parameters at times $t+1$ and t , $\bigtriangledown(Q(x_t, w_t))$ is the

gradient of loss function and α is the learning rate. Typical Loss functions minimised are MAE and RMSE. SGD is a preferred way to *fit* continuous data.

When output y_t is a linear combination of past values, the process is called *Auto Regression* (AR). In a *Moving Average* (MA) model, the output y_t is a linear combination of forecast errors. In *Auto Regression Moving Average* (ARMA) model, both AR (p) and MA (q) are used to predict y_t as shown in Equation 4.6 [121].

$$y'_t = \underbrace{c + \phi_1 y'_{t-1} + \phi_2 y'_{t-2} + ... + \phi_p y'_{t-p}}_{auto-regression} + \underbrace{\theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + ... + \theta_q \epsilon_{t-q} + \epsilon_t}_{moving-average} \quad (4.6)$$

Where order p captures the number of past values y future value of y depends on. y_{t-1},y_{t-2},...,y_{t-p} are previous observations of y, ϕ_1,ϕ_2,...,ϕ_p are parameters and order q captures the number of past values of ϵ, the future value of y depends on, ϵ_{t-1},ϵ_{t-2},...,ϵ_{t-q} are past errors and θ_1,θ_2,...,θ_q are parameters.

Values of p and q are calculated using ACF to identify correlation between captured data for batch learning. Since the entire dataset is unavailable for incremental learning, the value of p cannot be determined by using ACF plots.

4.5.3.3 Operation

TPagent treats network traffic as a continuous data stream while applying a simple linear mechanism of ARMA to predict traffic one step ahead. The agent does not use a large training dataset but starts predicting and learning with a single data point. To minimise computed loss, the agent calculates a simple rolling average using incoming data points and adjusts weights using SGD. Such operation allows the agent to learn *incrementally* while making immediate predictions based on historical observations.

The agent's operation is captured in Figure 4.11. The agent creates individual ARMA models for each link. Since each link is mapped to a unique model, different

model parameters such as p, and q can be customised for the individual link for better predictions. While PSagent agent periodically queries and gathers port statistics and updates the global knowledge base, the TPagent predicts the next value $ypred_{t+1}$ based on current reading $ytrue_t$ and past p readings. This process is repeated for all the links. In the next instance, $t+1$, the agent calculates loss over the received $ytrue_{t+1}$ and predicted value $ypred_{t+1}$ using the loss function MAE in Equation 4.7 to adjust weights of the model $\phi_1, \phi_2, .., \phi_p$ and $\theta_1, \theta_2, ..., \theta_q$.

$$MAE = 1/n(\sum_{i=1}^{n} ytrue_t - ypred_t) \tag{4.7}$$

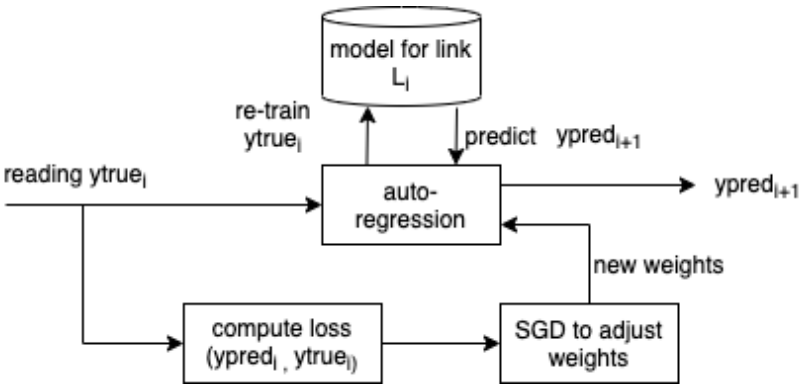


Figure 4.11: Traffic prediction agent

4.6 Management

Management layer agents work towards maintaining a desirable state. Network abnormalities in hardware - such as link and network device faults, physical link loops or software - such as unfair resource allocation, forwarding loops, non-optimal link utilisation, non-adherence to Service Level Agreements, and security breaches are undesirable. Rerouting, TCPFairness and PortDown agents are three agents that operate in the management layer.

Reroute agent (RRagent) rerouting flows to alternate paths to distribute traffic across the network. RRagent reasons flow allocation problem as a Constraint Satisfaction Problem. TCPFairness agent (TFagent) ensures fair bandwidth allocation amongst different TCP senders. This agent uses reinforcement learning to allocate bandwidth to TCP flows fairly. PortDown agent (PDagent) reroutes affected flows to alternate paths under no bandwidth constraints.

Management agents are autonomous, in the sense they constantly interact with the knowledge base and act upon identifying a network anomaly.

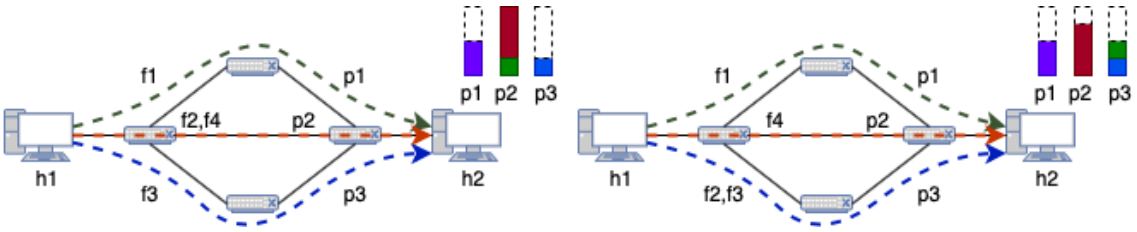
4.6.1 PortDown agent

Port status fluctuation triggers the datapath to send an OFPT_PORT_STATUS message to OFagent. Specifically, when a port status changes from *down* to *up*, the TOagent rediscovers the entire network. When a port status changes from *up* to *down*, this information is sent to the PDagent. PDagent removes the port and corresponding link information from the knowledge base and initiates the process of re-provisioning affected flows. KBagent identifies affected flows and IPagent reprovisions affected flows. PDagent doesnt wait for flow rules to timeout, and purges the knowledge base proactively.

4.6.2 Reroute agent

As previously described, agents in the provisioning layer provision a flow on the perceived best path at a given instance, as shown in Figure 4.12a. Nevertheless, since flow demands and duration vary, bandwidth can be further utilised efficiently by rerouting active flows.

In Figure 4.12a, active flows $f1$, $f2$ and $f3$ are provisioned on paths $p1$, $p2$, $p3$ respectively. New flow $f4$ is provisioned on the perceived best path $p2$ leading to congesting path *p2*. The desired behaviour would be as shown in Figure 4.12b where flow $f2$ is rerouted via $p3$ to free up resources for $f4$ on $p2$.



(a) pre-rerouting (b) post-rerouting

Figure 4.12: Active flow rerouting

The knowledge base provides congested links, and corresponding affected flows information to RRagent. A link is congested if the link load exceeds a set link threshold of 80%. It has been observed that a threshold of 70% causes frequent reroutes, and a threshold of 90% results in delayed or no rerouting. The agent chooses an arbitrary number of flows {F} of varying demands to reroute.

In the current setup, the agents choose these flows randomly. By choosing only high-demand flows to reroute, no path might exist to accommodate such high demands. On the other hand, if only flows of low demand are chosen, many such flows need to be rerouted to alleviate congestion. In either case, the agent would have to iterate over multiple sets of flows repeatedly to find a suitable set to reroute. Hence, a random selection allows agents to choose flows of varying demands.

4.6.2.1 Related Work

Weighted Cost Multipath Routing (WCMP) [123] distributes traffic amongst the available next-hop nodes in proportion to available link capacity.

Authors of [71] propose FlowBender that enables end-host to drive flow-level load balancing scheme by using Explicit Congestion Notification (ECN) for congestion detection and rerouting large flows. The solution uses the Equal Cost Multi-Path (ECMP) hashing mechanism to hash against a flexible field in the packet. This field is updated in case of congestion, thus allowing ECMP to handle packets differently.

In [124], authors propose an OFLoad scheme that separates elephant flows from mice flows and aims to minimise network congestion by routing the elephant flows on a single shortest path. Mice flows are aggregated and routed using WCMP.

[73] proposes Mahout, which introduces a Shim layer at the end-host to detect elephant flows. Switches use a packet's Differential Services (DS) Field to notify the controller of elephant flows. The controller places elephant flows on the best path.

In [65], authors aims to maximise aggregated network utilisation by dynamically scheduling flows.[125] proposes Niagara that uses wild card rules to split aggregate incoming traffic on the same set of next-hop nodes based on a weight vector.

In [72], weights are assigned to links, and new flows are configured on the path with the least weight. In [126], Dijkstra's algorithm finds multiple equal-length paths. In the event of congestion, higher priority flows are rerouted to links with the least cost and form the shortest path.

In [127], authors describe a network monitoring module that monitors the data plane every second. A load distribution module calculates the amount of load to be rerouted to backup paths.

Unlike earlier works, RRagent reassigns active flows using constraint solving as described in the following section.

4.6.2.2 Constraint-solving using backtracking and pruning

Classical search techniques find a solution by systematically searching through a solution tree. The agent backtracks upon reaching a leaf and navigates through new branches until a solution is found. Navigating all possible nodes is resource-consuming. Backtracking search [91] with constraint propagation is a systematic search mechanism where a partial solution is extended, and any assignment that fails to satisfy a constraint results in the pruning of the entire branch, and the algorithm backtracks to navigate and find alternate solutions.

Suppose $\{V\}$ is the set of variables and set $\{D\}$ is the domain of all possible values a variable can take. In that case, a solution is found if all variables in V have a legal value assigned from D. A legal value is a value that satisfies a set of constraints [91]. Backtracking search assigns a variable $v1$ a legal value $d1$ from $\{D\}$. The algorithm then assigns the next unassigned variable $v2$ a value $d2$. This process continues as long as constraints are satisfied, and $\{D\}$ has legal values to assign it to variable vi, and the search stops when a solution is found. Upon detection of an assignment that fails constraint or when a variable has no legal values for assignment, a branch is considered unsuitable and pruned entirely and allows the algorithm to backtrack early, as shown in Figure 4.13.

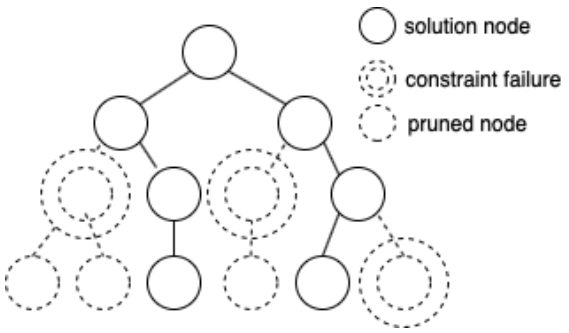


Figure 4.13: Back Tracking with Pruning

4.6.2.3 Operation

Displacement of congestion from one part of the network to a different part is a possible risk of rerouting. Not all paths can accommodate the new flow demands, and though some paths can accommodate a flow, provisioning the flow might congest the path for other active flows. This requirement is captured in the link-load constraint 4.6.2.3.

In addition to definitions of bandwidth capacity of link l bw_l, throughput th_l 4.1 (also referred to as link-load), available BW abw_l 4.3 established in section 4.5.2 following definitions aid in formulating rerouting problem:

- Flow: A flow f represents packets originating from source IP address (src_ip) towards destination IP address (ds_ip), port (pr) tuple.
- Path: If v is the set of all nodes in a fully connected network, and L is the set of all links, then a path p between source src and destination nodes dst is a subset of nodes v_p, sourcing at src and sinking at dst, connected via a subset of links L_p.
- Path-capacity: If $abw_{l1}, abw_{l2}, abw_{l3}...abw_{ln}$ are the available bandwidth along links $l1, l2, .., ln$ in path p. A path's available capacity pc_p is the lowest available bandwidth in the path p given by Formula 4.8

$$pc_p = min(abw_{l1}, abw_{l2}, abw_{l3}...abw_{ln}) \tag{4.8}$$

  Such a definition of path capacity will allow the algorithm to not provide a path with a flow beyond the capacity currently offered by the most congested link in the path.

- Path-load: If $th_{l1}, th_{l2}, th_{l3}...th_{ln}$ are current link throughput of a path p,then path-load pl_p of path p is given by Formula 4.9

$$pl_p = max(th_{l1}, th_{l2}, th_{l3}...th_{ln}) \tag{4.9}$$

- Flow-demand: Flow-demand fd_f of a flow f is current throughput of a flow. This information is queried from the datapath and calculated using Equation 4.10, where x is the number of bits matched against a flow rule and flow duration is the duration of the flow.

$$fd_f = x/flow - duration \tag{4.10}$$

Thus, the RRagent has to reroute a flow that satisfies Link-Load Constraint C1 4.6.2.3.

C1: A flow f of flow-demand fd can be provisioned on path p, if and only if there exists no link whose link-load (or throughput) exceeds link threshold due to action $pl_p + fd_f$. Such a constraint prevents the agent from displacing congestion from one link to another.

The RRagent performs constraint checking and backtracking to prune all solutions that do not satisfy constraint 4.6.2.3. Finding a solution begins with assigning a flow f of flow-demand fd_f to a path p with path load pl_p and extending this solution to configure other affected flows only if the link-load constraint 4.6.2.3 is satisfied. Since a link can be part of multiple paths used by other flows, the agent checks for constraint satisfaction for all the links and proceeds to extend this current solution cur_sol by provisioning remaining flows. If an assignment does not satisfy the constraint, the agent backtracks and assigns f to an alternative path and prunes all the subsequent flow assignments of the current branch. The process is repeated until all selected flows are assigned to a path. At this stage, the agent has a complete solution cur_sol. In Figure 4.14, the agent back-tracks at Step1 as assigning flow $f1$ ($fd_{f1} = 9$) to path $p1$ ($pl_{p1} = 8$) doesn't satisfy constraint C1 ($8 + 9 >$ threshold(16)). The agent backtracks without investigating further assigning along this branch and examines other assignments, as shown in Step 2.

Backtracking algorithm and corresponding constraint check mechanism employed by the agent as listed below:

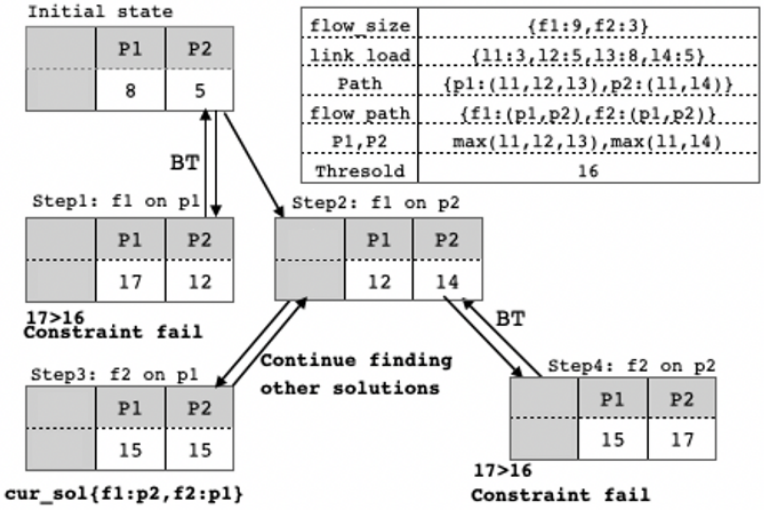


Figure 4.14: Example flow assignment

FIND-SOLUTIONS($cur\_sol, feasible\_set$)

```
1  f = select unassigned flow(F)
2  if f == None
3        feasible_set.append(cur_sol)
4        return feasible_set
5  recompute available_bandwidths()
6  if cur_sol[f] == None
7        for p ∈ paths
8              pl_p = pl_p + fd_f
9              cur_sol[f] = p
10             if CONSTRAINT-SATISFIED()
11                   FIND-SOLUTIONS(cur_sol, feasible_set)
12             cur_sol[f] = None
13             pl_p = pl_p − fd_f
14 return feasible_set
```

CONSTRAINT-SATISFIED()

```
1  for l ∈ links
2      if th_l > threshold_l
3          return False
4  return True
```

4.6.2.4 Cost-Function: Standard Deviation

A single search might not necessarily yield the best solution; a solution that minimises cost function is considered the best. The agent continues to search for other possible solutions for the best solution. RRagent aims to minimise the standard deviation of path-loads across a set of paths as a cost function.

COST-FUNCTION($feasible\_set$)

```
1  for sol ∈ feasible_set
2      for f ∈ sol
3          pl_p = pl_p + fd_f
4      sol_std = standard − deviation(th_l)
5      sol_val = sol_std/mean
6  return min(sol_val), sol
```

At this point, a solution is available for the agent to reroute active flows. Corresponding end-to-end flow rules are composed and configured on the data plane. The agent's operation is shown in Figure 4.15.

Since the cost function fluctuates as new flows are added to the path, it does not exhibit any increasing or decreasing trends, hence a non-monotonic function. Thus, solutions based on greedy and premature calculations of the cost function are rendered useless. Standard deviation amongst the path loads can only be calculated at the tree's leaves after all selected flows are assigned to paths.

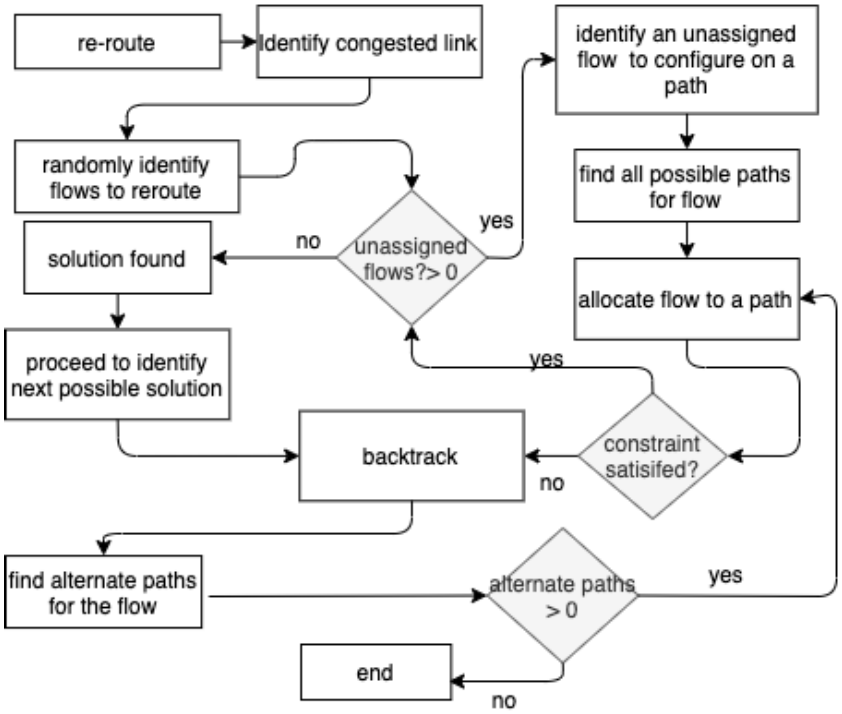


Figure 4.15: operation of rerouting agent

4.6.3 TCP Fairness agent

Transport Control Protocol (TCP) is a reliable, connection-oriented point-to-point transport protocol where end devices negotiate connection parameters during the initial handshake (also called a three-way handshake). TCP establishes session buffers during handshake and exchanges this information with connecting end device. Since TCP is full-duplex, end hosts on either side have a send buffer *sndbuf* and a receive buffer *rcvbuf*. Application data is captured by TCP and held in *sndbuf* at the sender node, and *rcvbuf* is used to hold data from the IP layer and read by the application at the receiver end. The sender tracks the sequence of bytes sent using a sequence number (SEQ). The receiver tracks received bytes using the ACK number. At the receiver end, *rwnd* is the free buffer space available before the receiver hits an overflow. *rwnd* is a variable and varies on factors such as application read rate. At any given instance, the sender must not send more bytes than can be held in *rwnd*.

The sender uses a variable *cwnd* to determine the number of bytes to send before

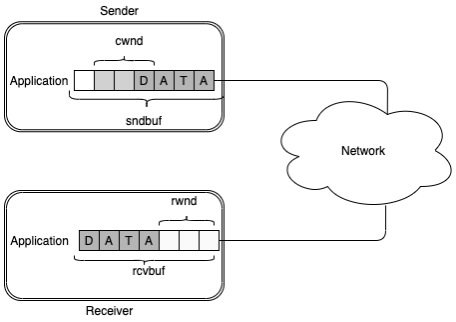


Figure 4.16: Sending and Receiving buffer

expecting an acknowledgement. At a minimum, it takes 1 Round Trip Time (RTT) for the sender to receive an Acknowledgement (ACK) from the receiver. Thus, the sender's sending rate for *cwnd* bytes of data is given by Formula 4.11 [128].

$$\frac{cwnd}{RTT} bytes/sec \tag{4.11}$$

and TCP throughput is given by Formula 4.12.

$$TCP_{throughput} = \frac{MSS}{RTT * \sqrt{(L)}} bytes/sec \tag{4.12}$$

where MSS is the maximum segment size, and L is the probability of packet loss.

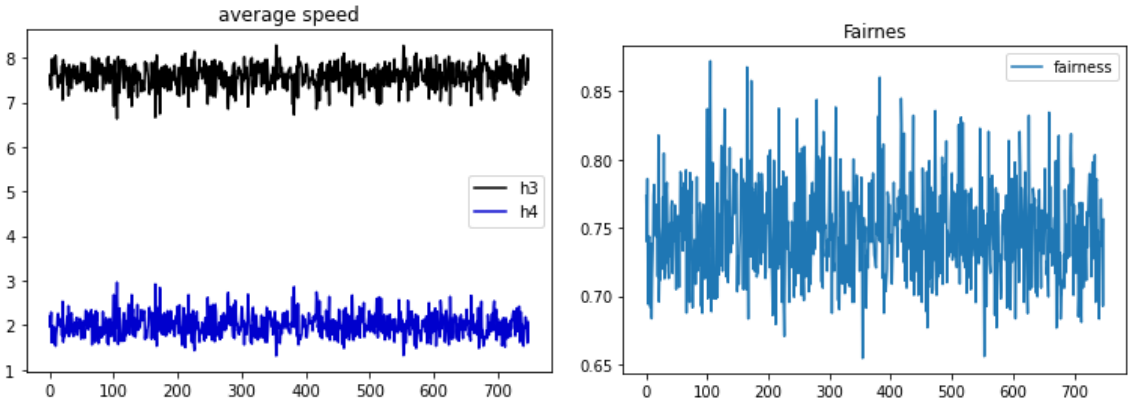
Flows with the same RTT and start time adjust their congestion window *cwnd* simultaneously according to TCP's Additive Increase Multiplicative Decrease (AIMD) mechanism, thus are allocated resources fairly. Typically, in such a setup - the rate of transmission of a flow is R/n where R is the rate of transmission of a link and n is the number of flows using a link. Such equal resource allocation for flows sharing a bottleneck link is ideal and of rare occurrence. Jain's fairness index Formula 4.13 [129] quantifies how fair TCP allocates bandwidth for the same link-sharing flows.

$$FairnessIndex = \frac{(\sum_{i=1}^{n} x_i)^2}{n * \sum_{i=1}^{n} x_i^2} \tag{4.13}$$

where x_i is the rate of transmission of a flow i, and n is the total number of flows

sharing the bottleneck link.

For flows with unequal RTT, flows with smaller RTT will receive acknowledgements early and increase *cwnd* faster than flows with higher RTT. Such flows with unequal RTTs sharing a bottleneck link are referred to as *heterogeneous flows* [130]. Since sending rate is directly proportional to *cwnd* and inversely proportional to RTT, a flow's throughput is inversely proportional to RTT. Higher RTT can result from network congestion or propagation delays; thus, flows with higher RTT are starved of TCP resources. In Figure 4.17, link speeds for hosts h3 and h4 are set to 10Mbps. Host h4 is set to experience a delay of 30ms while host h3 experiences a delay of 5ms. As is evident from Figure 4.17a, the transmission rate of h3 is higher than h4, and the fairness index varies between 0.65 and 0.85, as shown in Figure 4.17b.



(a) Throughput (b) Fairness Index

Figure 4.17: TCP Throughput and Fairness index for heterogeneous flows

4.6.3.1 Related Work

Traditionally, fairness in heterogeneous TCP flows was ensured by employing mechanisms such as Random Early Detection (RED) [131] to detect congestion and drop packets in advance before the occurrence of congestion. Early detection solves the TCP synchronisation problem amongst heterogeneous flows and ensures fairness. CHoKE proposed in [132], matches incoming packets with a randomly chosen packet from the First-In-First-Out (FIFO) queue and, upon a match, drops the packet. Both RED and CHOKe perform early detection to drop packets to penalise misbehaving flows.

With the advent of machine learning, new techniques to address congestion and fairness are being researched and investigated. [133] proposes QTCP based on NewReno TCP that applies reinforcement learning to vary congestion window *cwnd*. While network parameters such as sending interval, ACK interval and RTT are chosen to represent the network state, the agent varies *cwnd* and reaps the rewards based on higher throughput and low latency. Authors employ a model-free technique of Q_learning [97] to choose appropriate actions (increase 10 bytes, decrease 1 byte or not change) for the *cwnd* window during the congestion avoidance phase of new Reno.

[134] proposed two machine-learning models to address TCP congestion control. A loss Predictor agent (LP-TCP) is a supervised learning-based model with a learning agent, an actuator and a sensing engine. While the sensing engine senses ACKs and provides the learner with state information, the learner estimates the probability of current packet loss and provides this info to the actuator. Based on the probability of loss, the actuator decides if to send the packet. The second formulation of the problem is based on Reinforcement Learning (RL-TCP). In this formulation, the learner reaps a reward from the network and provides the actuator with action values to determine the best action to choose.

[135] propose AURORA. AURORA applies Deep Reinforcement learning to adjust sending rate. As for any reinforcement learning algorithm, the state information includes latency ratio, sending ratio and latency gradient. Actions that the agent is allowed to perform are adjusting sending rate, and the agent reaps the rewards as a function of throughput, latency and packet loss. [136] formulate TCP's congestion control mechanism as reinforcement learning based TCP-RL where the network's state snapshot is captured in throughput, RTT and loss rate. The agent adjusts the initial congestion window and the congestion control algorithm based on the state and reaps the rewards as a function of throughput and RTT.

While the earlier works propose reinforcement learning-based TCP congestion control mechanisms, the following works address TCP fairness amongst heterogeneous

TCP flows.

[137] proposes a mechanism to improve fairness achieved by QTCP by increasing the *cwnd* in the AIMD manner, unlike Multiplicative Increase Multiplicative Decrease (MIMD) manner of the original QTCP. [138] proposes Deep Q-Network (DQN) based congestion control mechanism to adjust *cwnd* while maintaining a higher utilisation rate. The agent considers link threshold, *cwnd*, RTT for state space, and action space includes adjustments to *cwnd* based on the probability of congestion. The agent gains reward based on *cwnd*, RTT.

The proposed TCP fairness agent in the MASDN control plane employs a reinforcement learning algorithm to adjust the flow's congestion window with the highest throughput to maintain fairness in resource allocation. Term *Fast* sender referes to such sender whose *cwnd* increases at a faster rate than a *slow* sender's *cwnd*. The reinforcement learning agent specifically learns to flag congestion to decrease the sending rate of the fast sender, thus allowing the slow sender to increase its congestion window gradually.

4.6.3.2 Reinforcement Learning

Reinforcement learning is a sub-branch of machine learning algorithms. As briefly mentioned in Chapter 3, machine learning algorithms are broadly classified into supervised, unsupervised, and reinforcement learning. In over-simplified terms, in supervised learning, a model is trained on datasets (inputs: X, outputs: Y) to learn function f such that $f(x) = y$ where $x \in X, y \in Y$. Upon encountering a new data point x_i the model plugs x_i in $f(x)$ and produces corresponding y_i' as output. This predicted value y_i' is compared with the actual output y_i to compute the loss of the model. Training the model with vast data sets reduces loss. Unsupervised learning algorithms handle unlabeled data by classifying and grouping them based on inferred patterns.

Reinforcement learning (RL) is online learning. That is, the agent does not

train on datasets. Instead, the agent explores the environment in which it operates, performs some actions and eventually learns what actions are beneficial. A typical reinforcement learning setup is a *Markov decision process* having a state (S), transition probabilities (T), possible action set (A) and a reward function (R). Figure 4.18 shows the interaction between RL-agent and environment.

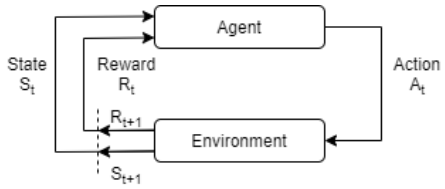


Figure 4.18: Reinforcement learning agent in action [139]

For the reader's convenience, below is a brief reproduction of definitions provided in [139]. For further reading, readers are directed to the same.

- *Reward* signal R_t formalizes the goal of the agent. *Expected return* (G_t) 4.14 is a function of these reward signals over time and γ is a discount factor of future rewards. The agent's purpose is then to maximise this return.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{4.14}$$

- *Policy* π is mapping between states and probability density functions of actions one can take in that state. $(a|s)$ is probability of taking an action a in a state s, policy $\pi(s)$ is an legal action a recommended by policy π in state s. A policy that yields maximum value is optimal π^*.

- *Value* function computes the worth of states and actions as functions of rewards. *Value function* of state *V(s)* is simply the expected return of following a policy π in a state s and is given by Bellman's equation 4.15 for v_π.

$$v_\pi(s) \doteq E_\pi[G_t|S_t = s] = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')], \forall s \in S \tag{4.15}$$

where $p(s', r|s, a)$ is the probability of landing in-state s' and reaping reward r after taking action a in state s.

Thus, the value function of a state is simply the probability of taking an action times immediate reward and future state's values, as visualised in Figure 4.19(a).

- *Value* function $q_\pi(s, a)$ of an action a under policy π in state s is given by Bellman's equation 4.16 for q_π.

$$q_\pi(s, a) \doteq E_\pi[G_t|S_t = s, A_t = a] = \sum_{s',r} p(s', r|s, a)[r + \gamma q_\pi(s')], \forall s \in S \quad (4.16)$$

Thus, the value function of action is the probability of landing in a state, times the immediate reward gained and the value of future actions. The term Q-value refers to the an action a's value function.

Slightly modified versions of backup diagrams provided [139] are visualised in Fig 4.19(b).

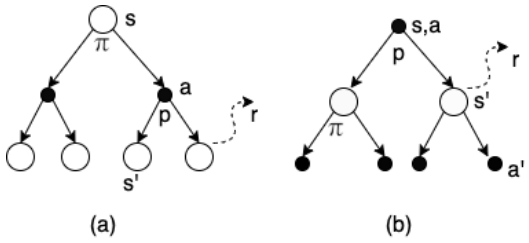


Figure 4.19: Backup diagram for v_π and q_π[139]

- *Transition model* mimics the changes in the environment. When a transition model is available, the agent knows the state it might land in upon acting. Only sometimes are transition models available to the agent.

Q-Learning

[140] coins the phrase *primitive learning* to refer to an agent that does not have or does not estimate a transition model or a reward function. Such learning is also known as model-less learning. Q-learning [140] is one such learning technique that requires no prerequisite information about transition models and reward functions. Instead, Q-learning is based on temporal difference prediction. The agent updates the estimate incrementally as a weighted sum of the existing Q-value and newly computed Q-value as in Equation 4.18.

$$Q(s,a) \leftarrow (1-\eta) * currentestimate + \eta * newestimate \tag{4.17}$$

that is,

$$Q(s,a) \leftarrow (1-\eta)Q(s,a) + \eta(r + \gamma max_a Q(s',a)) \tag{4.18}$$

where η is a constant $(0 < \eta \leq 1)$

The agent starts by exploring the environment by choosing a random action. Random actions allow the agent to *explore* the environment and to transition into as many new states as possible. Such discovery of new states and rewards is called an *experience*, and the Q-learning agent maintains a Q-table to store all the observations. An observation is a tuple of current state, action, immediate reward and next state (s,a,r,s'). Exploration allows agents to discover new state-action pairs but hinders agents from reaching the goal early. On the other hand, if the agent exploits learnt best actions of a state too prematurely, the agent eventually ignores less frequented state action pairs- thus learning inefficiently and landing in local maxima. Thus, most agents employ $\epsilon - greedy$ approach to balance exploration and exploitation. The agent chooses a random action with a probability of ϵ and the best action with a probability of $1 - \epsilon$. The best action has maximum Q_value $max_a Q(s',a)$.

Q-learning algorithm [140] for estimating policy π is as below

Q-Learning()

```
1  for ep in episodes
2      initialise s
3      for each_step in ep
4          choose an action a using ϵ-greedy algorithm
5          act and observe the next-state, reward
6          update q-value as Q(s,a) ← (1 − η)Q(s,a) + η(r + γmax_a Q(s′,a))
7          state ← next_state
8          if state == terminal:
9              break
```

Deep Q-Network

In discrete environments and environments with smaller state space, the agent stores policy ($\pi : S \rightarrow A$) in a Q-table. However, the tabular method becomes impractical in environments with large state or action spaces, such as continuous environments. Instead, a function approximator is used to generalise unknown states and reduce state size. This reduction is achieved by mapping sample state space to a small set of features.

While linear regression is one example of linear function approximation, function approximators can be non-linear such as a neural network. Deep Q-Network (DQN) proposed in [141], [142] employs a non-linear function approximator- a neural network with network weights (θ) as Q-network to predict Q_value $Q(s, a; \theta_i)$. Using SGD, the agent minimises the error $L_i(\theta_i)$ in Equation 4.19.

$$L_i(\theta_i) = E_{(s,a)\sim P(s,a)}[(y_i - Q(s, a; \theta_i))^2] \tag{4.19}$$

where

$$y_i = E_{s' \in S}[R(s, a) + \gamma max_{a'} Q(s', a'; \theta_{i-1})|s, a] \tag{4.20}$$

where $R(s, a)$ is reward for current state-action pair, and $Q(s', a'; \theta_{i-1})$ is Q-value of next state, action pair (s', a') using network parameters of earlier iteration (θ_{i-1}).

The authors of [141] employ *experience replay* to address the high correlation between successive observations where observations are stored in memory and randomly sampled to train the network to eliminate the correlation between sequential observations.

4.6.3.3 Operation

TCP allocates resources fairly when all the flows sharing a link have approximately equal throughput. Since TCP throughput is dependent on the congestion window (*cwnd*) and RTT of a flow, throughput varies significantly for flows sharing the same bottleneck link with different RTTs. For flows with higher RTT, the increase in *cwnd* is slower because ACK takes longer to reach the sender. Upon detecting congestion, TCP modifies *cwnd* for all the flows on a bottleneck link based on AIMD (additive increase multiplicative decrease) mechanism, adversely affecting slow senders.

Congestion is notified to a sender internally or explicitly. Both end devices and intermediate nodes can flag congestion explicitly using Explicit Congestion Notification (ECN). Intermediate devices flag congestion by turning on the congestion Encountered (CE) bit in IP header. Traditionally, an intermediate node with Active Queue Management (AQM) can only enable the CE bit. Upon receiving the CE flag-enabled IP segment, the receiver relays it back to the fast sender by setting the ECN-Echo bit in TCP header, triggering the sender to reduce its congestion window. TFagent does not use AQM, but instead installs flow rules to falsely flag congestion to force the fast sender to reduce sending rate.

TFagent adjusts *cwnd* of only those flows consuming high bandwidth (fast senders) while leaving low-bandwidth utilising flows (slow senders) to progressively increase their *cwnd* according to TCP's AIMD mechanism. To flag congestion, the agent configures a higher priority, hard-timeout flow rule on an intermediate switch. Thus

upon receiving a packet from the fast sender, the switch enables the IP header's CE flag, indicating congestion to the destination. The destination node enables the Explicit Congestion Echo (ECE) flag in TCP header to relay congestion notification back to the fast sender. As a response to the notification, the fast sender reduces the *cwnd*.

It has to be noted that random or continuous flagging of false congestion will result in the sender's *cwnd* reduced to a minimum, which is not desirable as this will reduce the overall throughput of the link.

As mentioned earlier, a reinforcement learning agent performs actions in a given environment to reap the rewards. Thus, TFagent's parameters are:

1. Environment's *state* is an observation made by the agent. Each observation is a list of the throughput $x_1, x_2..., x_n$ of all flows $f_1, f_2, .., f_n$ occupying a link and is of the form

$$s = [x_1, x_2, x_3...x_n]$$

2. Agent's *actions* are a list of possible actions at the agent's disposal. TFagent has three actions to choose from.

   - by choosing action *0* agent does not act on the current state of the environment.
   - by choosing action *1* agent enables CE flag for a flow with low flow demand. This is generally a bad action as it might lead to further reduced resources allocated to this flow.
   - by choosing action *2* agent enables the CE flag for a flow with high demand. This is a desirable action as it leads to the slowing down fast transmitters while other slow transmitters additively increase their *cwnd*.

3. Agent's rewards are obtained from a reward function. An agent is not explicitly aware of effect of its action's consequences. Actions can only be quantified based on their consequences in the environment and a reward function qualities

these consequences. Since, the goal of the agent is to increase fairness and maintain a healthy aggregate link throughput, the agent's reward function is based on fairness and the link's aggregate throughput. The agent reward function is given below.

$$rew = \begin{cases} +5, & fairness > 0.9 \\ +1, & fairness > 0.85 \\ -1, & aggr.th < 0.6 \\ 0, & otherwise \end{cases}$$

Where fairness is Jain's fairness index

$$fairness = \frac{(\sum_{i=1}^{n} x_i)^2}{n * \sum_{i=1}^{n} x_i^2}$$

And aggregated throughput is

$$aggr.th = \sum_{i=1}^{n} x_i$$

If the agent's action results in high fairness, a reward of 5 is gained. The agent gains a reward of 1 for maintaining fairness. The agent is punished if the aggregated throughput is below 60%, ensuring that the agent does not operate inefficiently while only maintaining fairness. Such punishment also ensures that the agent does not only aim to reap high rewards by constantly flagging congestion. The agent reaps no rewards otherwise.

Unlike supervised or unsupervised machine learning, reinforcement learning is incremental and occurs episodically or continuously. In each episode, the DQN-based TFagent observes the environment, an array of current flow sending rates. Observations are made at the source switches of each flow. Agent balances between exploration and exploitation by starting with extensive exploration and eventually tending towards exploiting actions. This transition is controlled by a parameter (exploration-rate ϵ).

During initial episodes, ϵ is high, allowing the agent to explore and choose random actions. As training episodes progress, ϵ reduces, and the agent chooses actions with better rewards. Selecting action 1, 2 translates to the configuration of the flow rule at the terminal switch. The agent waits for 1 sec allowing *cwnd* change to take effect before observing the environment. Once observed, the agent's neural network (Q-network) updates the Q-value of the action. This process continues for the entirety of the episode, as shown in Figure 4.20.

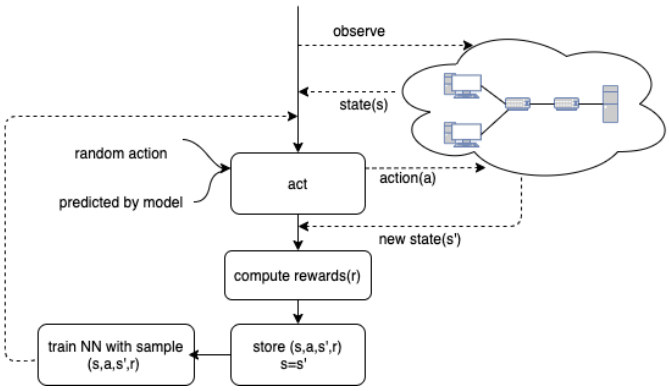


Figure 4.20: Operation of TCP fairness agent

4.7 Conclusion

This chapter presents an agent-based SDN controller architecture, and each agent's functionality is described. The operation of simple agents for topology discovery, provisioning flows, and collecting network statistics is presented. Three complex agents possessing various reasoning and learning capabilities have been discussed. RRagent reroutes active flows onto alternate paths under bandwidth constraints, TFagent ensures TCP fairness by learning to slow the fast sender and maintain high throughput temporarily.TPagent agent incrementally learns to predict traffic. The next chapter discusses the knowledge base used by the agent system and the agent system's communication patterns.

Chapter 5

Knowledge Representation and Communication

5.1 Introduction

As introduced in Chapter 3, agents use available knowledge to make decisions and knowledge is formally represented in logical languages such as Knowledge Interchangable Format (KIF) [98], First Order Language (FOL) [143], or in HTML type Extensible MarkUp Language (XML) [99], Resource Description Framework (RDF) [144] or in Web Ontology Language (OWL) [145] [92]. An ontology of a domain is a formal specification of concepts and relations amongst them in that domain while defining vocabulary for interaction and implementation [146]. Formal knowledge representation also allows automated reasoning, which requires elaborate domain representation. Reasoning is a process of deducing new facts from existing facts. In a traditional programming approach (e.g., object-oriented programming), a program handles facts based on the domain-specific logic programmed into functions or methods and cannot deduce new facts [91].

Agents in a multi-agent system interact and use global knowledge described in an ontology. Reasoners such as HermiT [147] infer new facts and inconsistencies based on this ontology. While the ontology presented in this chapter is sufficient for the

proposed agent system, it is not claimed to be complete. Instead, this chapter aims to provide a reference for building a further detailed SDN Knowledge base (KB). Ontology for the agent-based controller is defined in OWL standardised by the W3C working group.

5.2 OWL for ontology

This section briefly introduces OWL ontology building language and associated reasoner HermiT [148].

5.2.1 OWL language

In OWL, *Classes* represent concepts and *roles* represent properties. Classes can be related. A class might also be a subclass of (*owl:subclassOf*), or equivalent to (*owl:equivalentOf*) or disjoint of (*owl:disjointOf*) another class. A subclass member is automatically inferred as a member of the parent class. Two classes can be equivalent to each other. Disjoint classes do not share members. These relations are essential in OWL, where an individual can belong to multiple classes. All things (classes, individuals and properties) are sub-classes of a default *owl: Thing* class. *owl: Nothing* is another default class with no instances.

Class constructors allow the creation of complex or composite classes from simple classes. OWL provides intersection (*owl:intersectionOf*), union (*owl:unionOf*) and complement of (*owl:complementOf*) logical operators to construct complex classes. An intersection of two classes is a collection of objects belonging to both classes. A union of two classes is a collection of objects belonging to either of the classes. Finally, a complement class consists of objects that do not exist in another class.

Roles are properties of a class. An *owl:ObjectProperty* links two classes. These classes are called *domain* to capture the *subject* and *range* to capture the *object* in a

statement. Thus an object property is of the form:

$$objproperty(domain, range)$$

owl:DatatypeProperty links data (values) to the *domain*. The difference is *range* is now a data value - such as a decimal, float or a string. Data property is of the form:

$$dataproperty(domain, value)$$

Like class constructs, *role restrictions* enforce logical restrictions on various class relations. *owl:allValuesFrom* functions as a universal quantifier $\forall$ whereas *owl:someValuesFrom* functions as an existential quantifier $\exists$. Numerical restrictions such as *owl:hasValue* assign values to a datatype role. As a rule of thumb, sub-classes have an *is-a* relation with the main class, and roles have a *has-a* relation with classes.

Instances of a class are called individuals. Like classes, individuals have relations with other individuals. *owl:DifferentFrom* relation establishes that two individuals are different from one another. Two individuals are the same if they are related by *owl:sameAs* relation.

Figure 5.1 presents an example ontology. Class *Person* has two sub-classes - *Professor* and *Student*. Also, the Professor class is disjoint with the Student class, implying that a person cannot be both Professor and student.

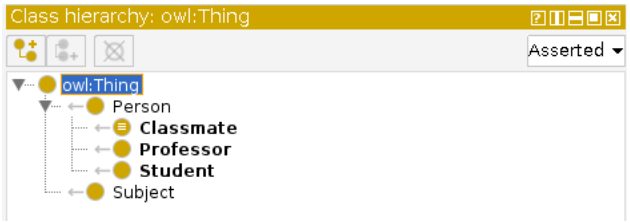
Protégé-2000 [149] is a graphical tool for representing and building OWL ontologies. In this thesis, ontology for the agent system is demonstrated using Protégé.

Professor and Student classes thus created are shown in Figure 5.2a, and class hierarchy is shown in Figure 5.2b.

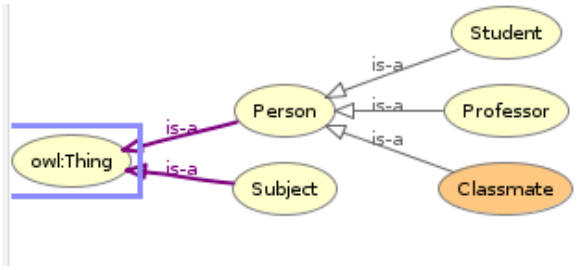
A statement such as 'Professor X teaches Subject C101', and 'Student A takes Subject C101' can be stated using object properties. In the statement 'Professor X teaches Subject C101' while Professor X and Subject C101 are class instances, *teaches* is an object property that establishes a relation between Professor X and Subject

```
    <owl:{Class}
        RDF:about="http://www.semanticweb.org/root/ontologies/2022/4/untitled-ontology-25\#
{Person}"/>
    </owl:Class>

    <owl:{Class}
        RDF:about="http://www.semanticweb.org/root/ontologies/2022/4/untitled-ontology-25\#
{Professor}">
        <rdfs:{subClassOf}
RDF:resource="http://www.semanticweb.org/root/ontologies/2022/4/untitled-ontology-25\#
{Person}"/>
        <owl:disjointWith
            RDF:resource="http://www.semanticweb.org/root/ontologies/2022/4/untitled-ontology-
25\#{Student}"/>
    </owl:Class>

    <owl:{Class}
        RDF:about="http://www.semanticweb.org/root/ontologies/2022/4/untitled-ontology-25\#
{Student}">
        <rdfs:{subClassOf}
        RDF:resource="http://www.semanticweb.org/root/ontologies/2022/4/untitled-ontology-25\#
{Person}"/>
    </owl:Class>
```

Figure 5.1: Ontology for Classroom



(a) Classes

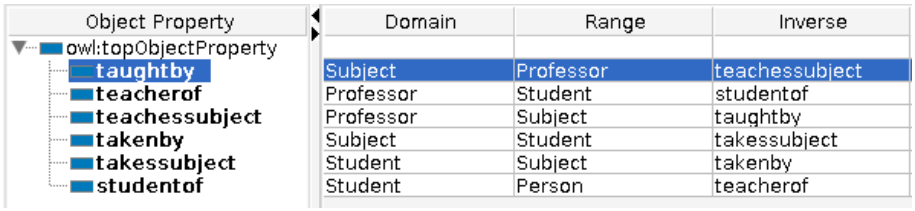


(b) Class Hierarchy

Figure 5.2: Classes in OWL

C101. Similarly, students have student IDs for individual identification. Relations between class instances and data types (integers, strings) are captured using data properties. *hasID* is a data property that assigns a student ID to a student. Object properties and data properties for the Example 5.2a are listed in Figure 5.3a and 5.3b, respectively.

Composite statements are made of two or more statements. For example, the



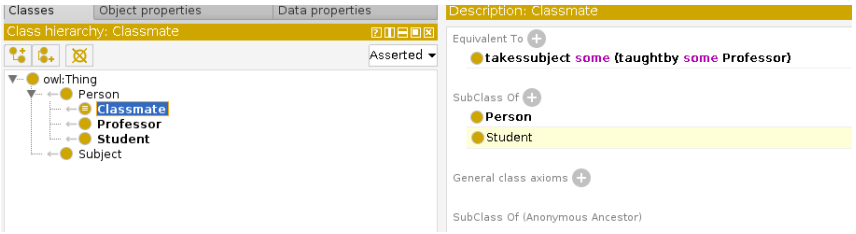
(a) Object properties

(b) Data properties

Figure 5.3: Roles in OWL

statement 'Student A is taught by Professor X' breaks down as 'Student A takes Subject C101' and 'Subject C101 is taughtby Professor X'. *Chaining* of properties expresses composite statements. Similarly, the classmate class can be expressed as a relation between two students who attend the same Professor's class. Classmates class is equivalent to group all students who take *some* subject taught by Professor. *Some* is a class restriction here. These composite statements are shown in Figure 5.4a and Figure 5.4b, respectively.

(a) Chaining properties



(b) Equivalent classes

Figure 5.4: Roles restrictions

5.2.2 Reasoning

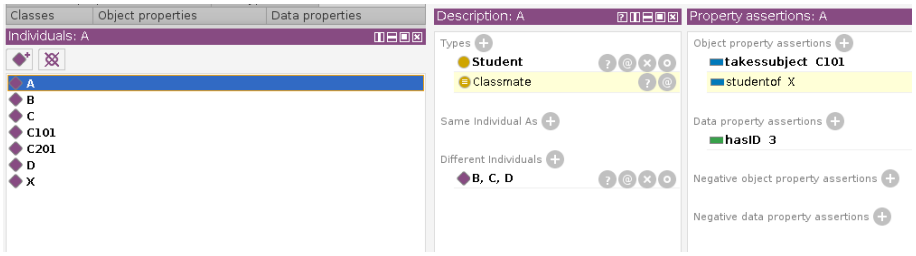
Work in [148] propose HermiT ontology reasoner. HermiT reasoner supports OWL 2 ontology language, reasons and identifies inconsistencies in the data presented. The main components of HermiT reasoner classify roles (Classification), chain roles (Loading), and manage instances (Realisation). Unlike other reasoners such as Pellet [150], HermiT uses *hypertableau* calculus [151]. HermiT represents ontology as Descriptive Logic (DL) [152]. Like OWL, DL works with individuals, concepts and roles. Indeed OWL is based on DL. DL represents the knowledge base as a set of assertions (ABox) that stores all named individuals and a set of terminology (TBox) to store complex descriptions of concepts and roles. Like OWL, DL allows universal $\forall$, existential $\exists$, unionof $\sqcup$, intersectionof $\sqcap$ relationships. The reasoner component in HermiT reconstructs OWL ontology into a set of assertions A and *DL-clauses*. A DL-clause is of form 5.1 [148], where B_i and H_i are atomic statements constructed from ontology.

$$B_1 \wedge ... \wedge B_n \rightarrow H_1 \vee ...H_m \tag{5.1}$$

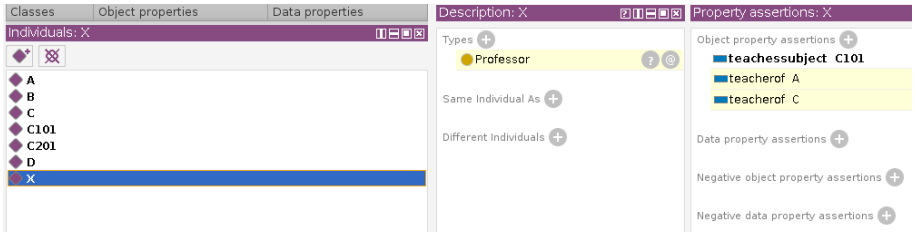
The reasoner solves the above FOL clauses to infer new facts and identify inconsistencies.

Continuing the earlier example, based on the information that Students A and B takes subject C101, the reasoner deduced that C101 is a subject taught by Professor X, and Professor X teaches Student A and Student B. The reasoner also inferred that since Professor X teaches Students A and B, A and B are classmates, as seen in Figure 5.6b. The reasoner derives new facts based on existing facts- which non-logical programming languages are incapable of.

Inferred information is highlighted in yellow.

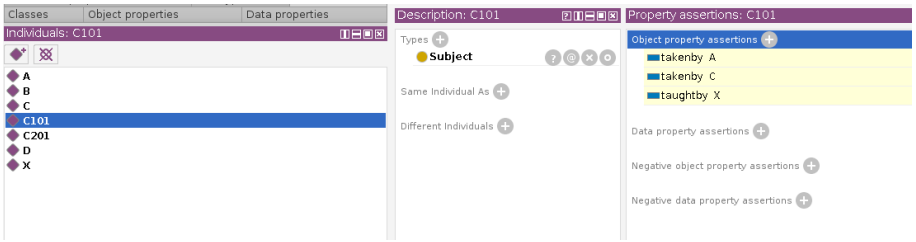


(a) Student A

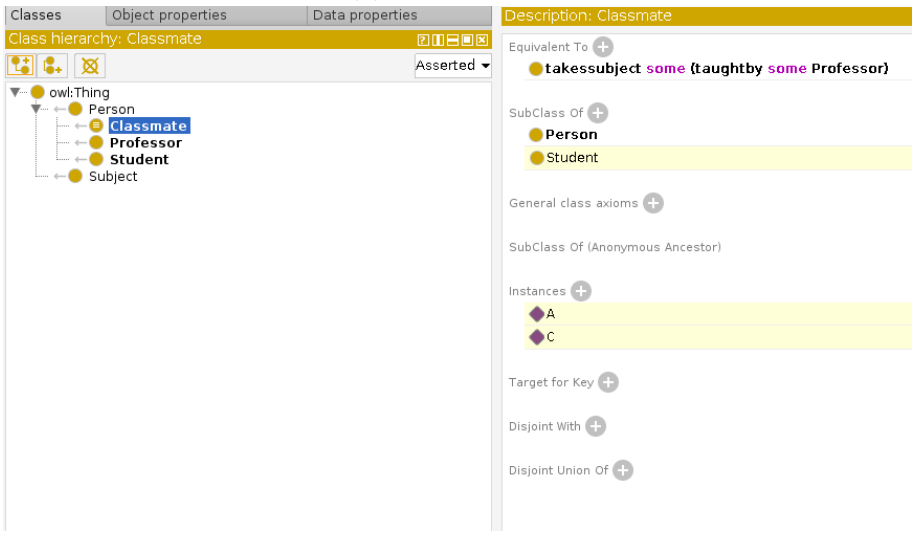


(b) Professor X

Figure 5.5: Roles restrictions



(a) Subject C101



(b) Classmates

Figure 5.6: Inferences

5.3 Ontology for Agent-based controller

It is now established that inference relies on the formal representation of knowledge. In this section, various entities of ontology designed for the multi-agent system are introduced. Necessary classes, roles and restrictions to capture information used by agents in the agent system are described. Since the concepts are not specific to the agent system, the ontology is extendable to include other network functions related to management and security. That said, the current ontology version captures information about switches, links, ports, topology, flows, flow rules and network abnormalities such as port down events and network congestion. A high class-level view of the ontology is presented in Figure 5.7.

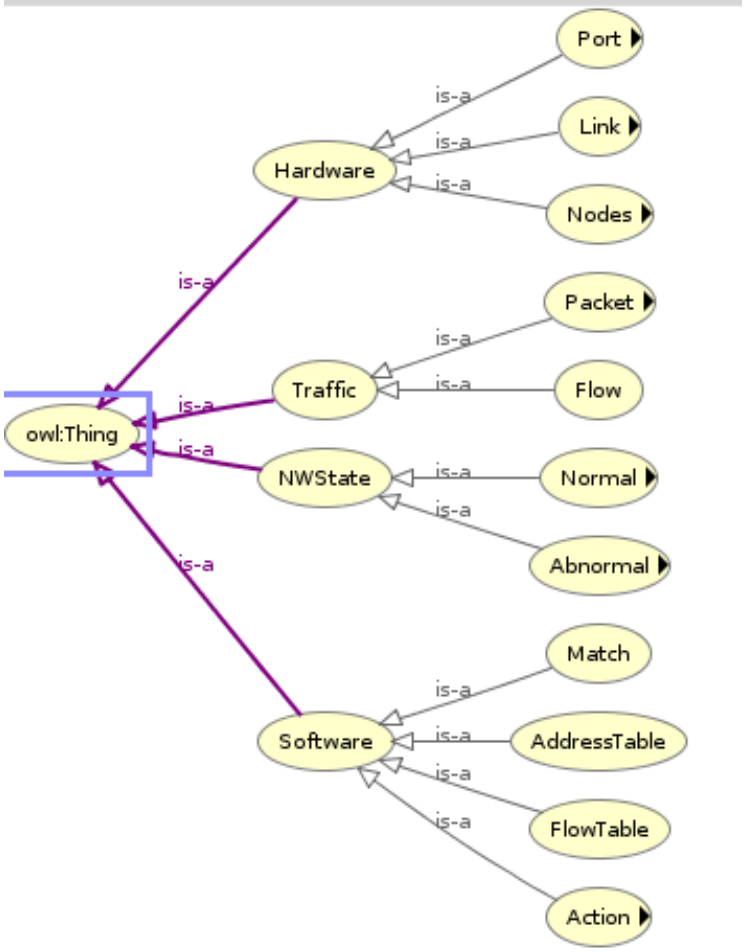


Figure 5.7: Classes in SDN ontology

Navigating from the root node *owl: Thing*, concepts captured by first-level classes are as below

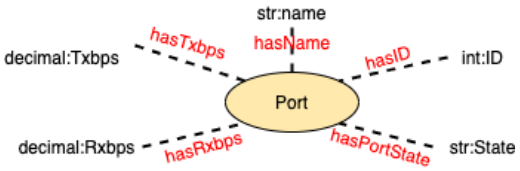
- *Hardware* - broadly captures concepts for physical layer. This includes switches, hosts, links and ports.
  - *Port* subclass captures the concept of hardware and software ports associated with hosts and datapaths. Every port has an ID and name for identification. Counters (TxBytes, RxBytes) keep track of the number of bytes transmitted and received by the port, and the rate of transfer in *bps* is captured in Txbps and Rxbps data properties. Port state can be one of the three states - *up*, *down* and *unknown*. While the *up* state indicates the normal functioning of the port, the *down* state indicates a port whose status is down due to possible abnormality. Unknown states indicate a non-readable port state. Figure 5.8a summarises data properties of the port class.
  - *Link* subclass captures the concept of media (e.g. a LAN cable) connecting a pair of nodes. A node is either a host or a datapath, and a link connects a pair of nodes. The current definition of the link represents uni-directional links. For example, *l14* represents a link from datapath dp1 to datapath dp4, and *l41* represents a link from datapath dp4 to datapath dp1. Link subclass has an ID and name for identification. The link state is captured as *up*, *down* and *unknown*. Like port status definitions, *up* is considered a normal operational state, and *down* states indicate a port whose status transitioned from *up* to *down*. Down states indicate a port abnormality (physical or operational). Link state is influenced by port state and vice-versa as when a port's status is registered as *down*, the corresponding link's status is inferred as *down*, maintaining consistency between port and link statuses. Figure 5.8b summarises data attributes for the link class.

– *Datapath* subclass is a high-level subclass to capture the behaviour of an OpenFlow switch. Datapath has ID, name and status as datatype properties. Directly connected datapaths and neighbours are inferred using object properties described in the upcoming section.

– *Host* subclass identifies an end device with name, IP, and MAC address as datatype properties. Hosts are connected to datapaths.

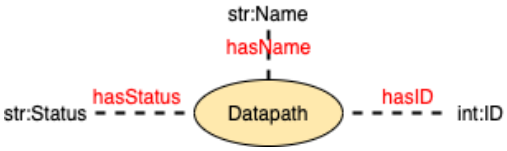
Data attributes for datapath and host classes are summarised in Figure 5.8c and Figure 5.8d, respectively.

Subclasses of hardware class and their corresponding properties are shown in Figure 5.8.

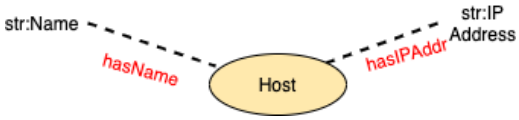


(a) Data attributes of port

(b) Data attributes of link



(c) Data attributes of datapath



(d) Data attributes of host

Figure 5.8: Sub-classes of hardware class and corresponding properties

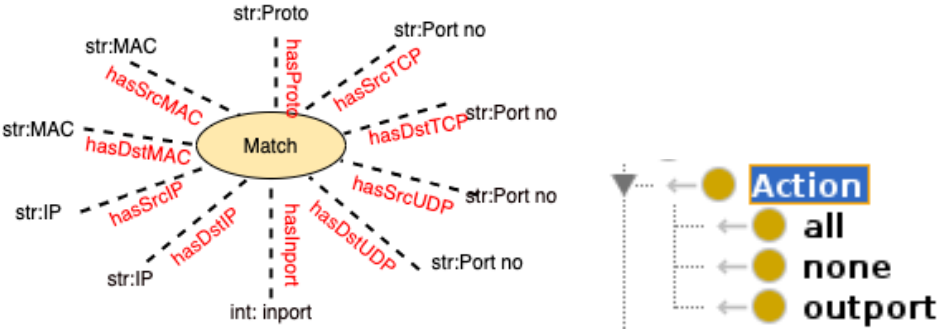
- *Software* - while the hardware class captures information of all physical components, the software class groups and captures the software configurations of these devices. Agent system's software classes are defined and classified as below:

  – *Match* subclass captures match criteria information used to compose a flow rule. Match sub-class's datatype properties are source and destination MAC addresses, source and destination IP addresses, Ethernet protocol and source, destination TCP ports for TCP flows and source and source and destination UDP port numbers and incoming port information.

– *Action* subclass has three subclasses. *all* action refers to flooding on all ports, an *outport* action refers to a unicast, and a *none* action indicates a packet drop action.

– *FlowTable* subclass stores flow rules similar to a OpenFlow datapath's flow table. As per the definition of flow rule, this subclass has two components - Match criteria to match the flows against, instructions to store action sequences to perform on flows.

– *AddressTable* subclass combines functions of an Address Resolution Protocol (ARP) table and a MAC address table. It represents a host's IP address, MAC address and connected switch information.



(a) Data attributes of match class (b) Sub-classes of action class

Figure 5.9: Sub-classes of software class and corresponding properties

- *Traffic* sub-class represents network traffic at two levels of abstraction.

  – *Packet* subclass represents a generic network packet. The packet class is sub-classed into L2, L3 and L4 packets based on the packet header.

    * L2 subclass represents L2 datagram such as Ethernet. As with Ethernet packets, individuals in this subclass have a source MAC address, destination MAC address and EtherType as data properties.

    * L3 subclass represents L3 packets. The L3 packet header has the source IP address, destination IP address, and IP protocol information captured in data properties. The IPv4 packet is a subclass of the L3

subclass. As with the L2 subclass, this can be extended to include other L3 packets, such as ICMP or IPV6 header information.

* L4 subclass represents L4 packets. The L4 packet header has the source port address and destination port address information. The L4 subclass has TCP and UDP subclasses.

– *Flow* subclass captures packet information at a high-level.

- *NWState*- sub-class identifies network state as normal or abnormal. Currently, two abnormalities are captured and represented in the abnormal class. The instances of the abnormal class are inferred and not pre-defined. The individuals are reclassified as instances of an abnormal class. This reclassification is based on the properties of an individual. Thus, two sub-classes of NWstate class are shown in Figure 5.10.

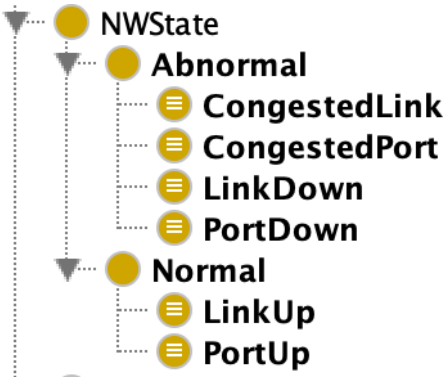


Figure 5.10: Network states

– *Normal* subclass represents a fully operational network. This is the default state of a network.

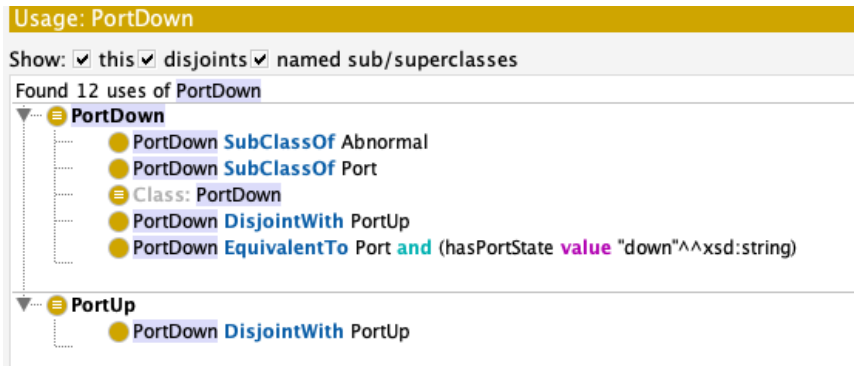
* *PortUp* subclass represents all the individuals of the Port class with the state *up*. This subclass is disjoint from the PortDown subclass.

* *LinkUp* subclass represents individuals of the link class connecting two ports in *up* state.

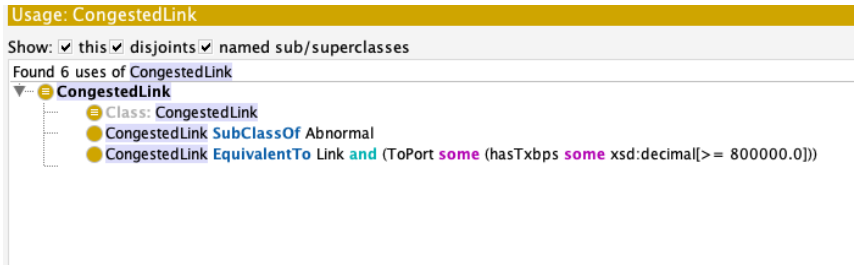
– *Abnormal* subclass represents a network state considered not normal. There are two types of abnormal states captured in sub-classes.

* *PortDown* subclass represents all the individuals of the Port class with the state *down*.
* *CongestedPort* subclass represents individuals of the port class with port counters exceeding a threshold and ports are classifies as congested.

CongestedLink subclass is a link that connects to a congested port similarly, a *LinkDown* identifies a link whose at least one connecting port is in *down* state. PortUp and PortDown classes are subclasses of both the NWState class and the Port class. They are disjoint classes and capture that a port cannot simultaneously be in both *up* and *down* states. Figure 5.11 summarises abnormal port state definitions.



(a) PortDown subclass



(b) CongestedPort subclass

Figure 5.11: PortDown and CongestedPort class definitions

Table 5.2 summarises definitions of all data properties, their domains and ranges.

Representing the concept of neighbourhood

Data properties are data attributes of classes, whereas *object properties* establish a relation between classes. For example, switch and port are distinct classes. A

Data property	Domain	Range
hasFlowRate	Flow	xsd:float
hasProto	Match or L3 or L4	xsd:integer
hasRXBytes	Port	xsd:integer
hasTxbps	Port	xsd:decimal
hasRxbps	Port	xsd:decimal
hasTxBytes	Port	xsd:integer
hasDstMAC	Match or L2 or L3 or L4	xsd:string
hasSrcMAC	Match or L2 or L3 or L4	xsd:string
hasFlowDemand	Flow	xsd:float
hasDstUDP	Match or L4	xsd:integer
hasSrcUDP	Match or L2 or L3 or L4	xsd:integer
hasDstTCP	Match or L2 or L3 or L4	xsd:integer
hasSrcTCP	Match or L2 or L3 or L4	xsd:integer
hasDstIP	Match or L3 or L4	xsd:string
hasSrcIP	Match or L3 or L4	xsd:string
hasID	Datapath or Host or Link or Port	xsd:integer
hasIPAddr	Host	xsd:string
hasPortState	Port	xsd:string
hasEtherType	Match or L2 or L3 or L4	xsd:string
hasInPort	Match	xsd:integer
hasMACAddr	Host	xsd:string
hasflowDuration	Flow	xsd:float
hasName	Datapath or Host or Link or Port	xsd:string

Table 5.1: Data properties

statement - 'port belongs to switch' establishes a *belongs to* relation between port and switch. Here Port class is *domain*, and the switch class is *range*. Interconnected or *chained* simple properties give rise to *composite properties*. Chaining properties result in transitive behaviour. Consider the topology in Figure 5.12a to elaborate further. Dp1, dp2, dp3, dp4 and dp5 are individuals of the datapath class. Dp1pr1,

dp1pr2, dp1pr3, dp1pr4 are individuals of port class representing ports of dp1 and l14, l15 are individuals of link class identifying links connecting dp1 to dp4 and dp1 to dp5 respectively

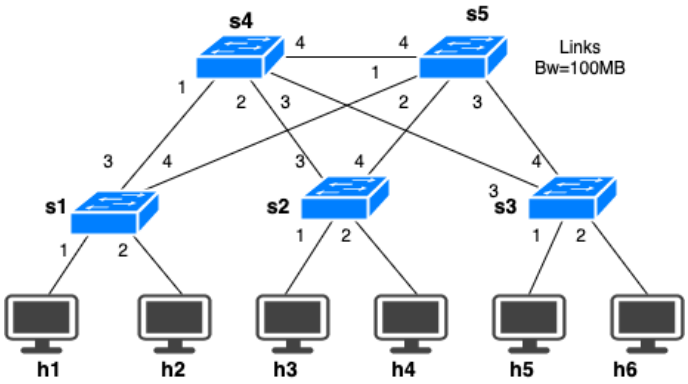
Consider datapath dp1. The concept of a port belonging to a switch is established using *hasPort* (domain: Node, Range: Ports) object property resulting in (dp1 *hasPort* dp1pr1, dp1 *hasPort* dp1pr2 , dp1 *hasPort* dp1pr3, dp1 *hasPort* dp1pr4) relations. In a physical network, media connects to ports. The concept is expressed similarly using *hasLink* (domain: Ports, Range: Links) object property resulting in (dp1pr3 *hasLink* l14, dp1pr5 *hasLink* l15) object properties. Such definitions are extended to other datapaths in the ontology.

Inverse properties define inverse relations. For example, *PortOf* (domain: Ports, Range: Nodes) is an inverse property of *hasPort* (domain: Nodes, Range: Ports) and *ToPort* (domain: Links, Range: Ports) is an inverse property of *hasLink* (domain: Ports, Range: Links) property.

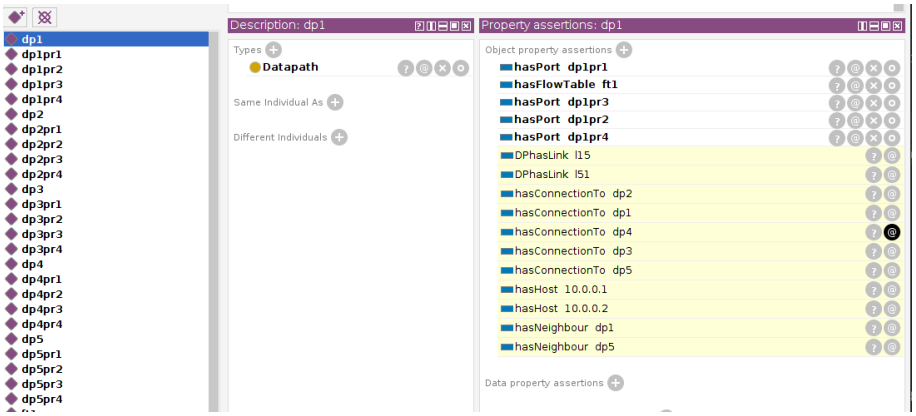
By chaining *hasPort*, *hasLink* and corresponding inverse properties, *neighbour* relation *hasNeighbour* (domain: Node , range: Node) is established between dp1, dp4 and dp1, dp5. The rule 5.2 expresses the neighbour relationship between two physically connected data paths by chaining multiple properties. To establish dp1 and dp5 as neighbours, datapath dp1 *has a port* dp1pr4 that *has a link* l15. By inverse property, link l15 *is connected to port* dp5pr1, which is a *port of* datapath dp5.

$$\textbf{hasPort} \circ \textbf{hasLink} \circ \textbf{ToPort} \circ \textbf{PortOf} \rightarrow \textbf{hasNeighbour} \tag{5.2}$$

The rule, 5.3 chains neighbours to define the connection to other datapaths. For example, dp1 is not a neighbour of dp3 as they do not share a link. However, datapath dp1 *has a neighbour* dp5, which, inturn, *has a neighbour* dp3. Thus, dp1



(a) Topology



(b) Neighbours and Connectivity

Figure 5.12: Defining neighbour relations

and dp3 are connected.

$$\textbf{hasNeighbour} \circ \textbf{hasNeighbour} \rightarrow \textbf{hasConnectionTo} \tag{5.3}$$

In Figure 5.12b, we see that not only is dp5 inferred as dp1's neighbour, but links l15 and l51 are also identified as connected to dp1. Similarly, dp1's connectivity to other datapaths is also established using inference.

Usually, such information is statically programmed in an object-oriented program as data structures or object properties. The program cannot infer that dp1 and dp5 are neighbours as they share the same link. Instead, the developer programs a logic to check that if a pair of nodes share a link, they are neighbours.

Representing a port down event

A normally functioning port has *hasPortState* data property set to *up* as shown in rule 5.4 whereas a malfunctioning port's state is set to *down* as shown in rule 5.5. Rule 5.4 puts a restriction that an individual of the Port class, with data property *hasPortState* set to *up*, belongs to PortUp class. Rule 5.5 puts a restriction that an individual of the Port class, with data property *hasPortState* set to *down*, belongs to PortDown class.

By defining PortUp and PortDown classes as disjoint and placing restrictions listed in rules 5.4 and 5.5, respectively, we ensure that a port is not identified as both *up* and *down* simultaneously.

Port and (hasPortState value "up"^^xsd:string) (5.4)

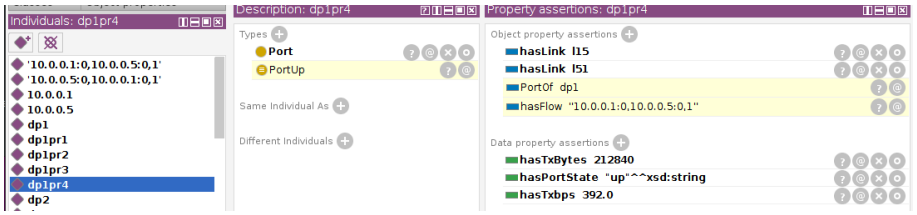
Port and (hasPortState value "down"^^xsd:string) (5.5)

A link is identified to malfunction if at least one connecting port is in *down* state. This restriction is captured in rule 5.6. The restriction states that an individual of link class, connected *toPort* some port whose *hasPortState* attribute is *down*, is classified as an instance of LinkDown class.

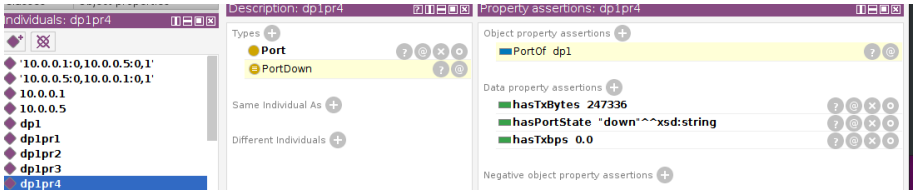
link and (ToPort min 1 (hasPortState value "down"^^xsd:string)) (5.6)

A port in *up* state is an instance of PortUp class. When the port's state transitions to *down* state, it is no longer classified as a member of PortUp class; rather, the reasoner reclassifies the port as a member of PortDown class. Thus, all the associations of the previous class, including flows, and links are removed. This transition is shown in Figure 5.13a and Figure 5.13b, respectively. In Figure 5.13a

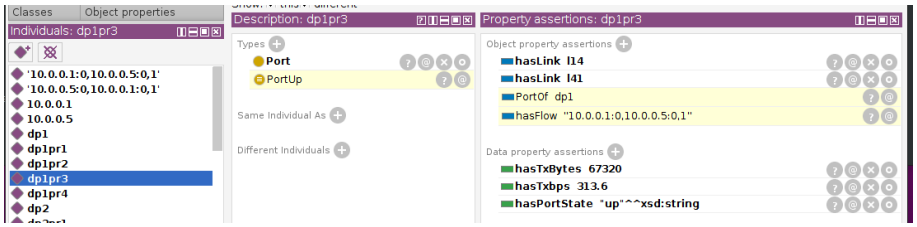
the port dp1pr4 of datapath dp1 is operational and connects to datapath dp5 via link l15 and used by flow *10.0.0.1:0,10.0.0.5:0,1*. In Figure 5.13b, we see that flows no longer use this port, and the port does not connect to any links while in *down* state. The flow has also been reassigned to use link l14 connecting node dp1 to dp4 as seen in Figure 5.13c.



(a) dp1pr4 port up



(b) dp1pr4 port down



(c) dp1pr4 port up

Figure 5.13: Abnormality 1 - Change of port state from UP to DOWN

Defining network congestion

Like the PortDown class, a different class (CongestedPort) is created and used to identify congested ports. A port is reclassified as *CongestedPort* if it is operating at 80% of the transmission rate. Rule 5.7 places restrictions for classifying a port as a member of *CongestedPort*. The restriction states that any port that *hasTxbps* value greater than 800000.0 is a congested port. Similarly, rule 5.8 states that any link connected *to any port* whose *hasTxbps* value greater than 800000.0 is a member

CongestedLink class.

$$\textbf{Port and (hasTxbps some xsd:decimal[>= 800000.0])} \tag{5.7}$$

$$\textbf{Link and (ToPort some (hasTxbps some xsd:decimal[>=800000.0]))} \tag{5.8}$$

Figure 5.14 shows an example of CongestedPort *dp3pr1*.

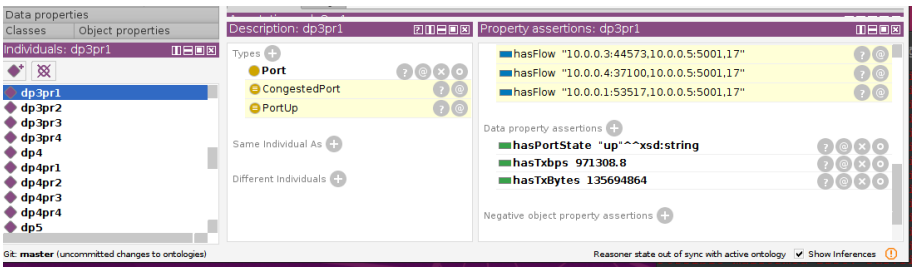


Figure 5.14: Abnormality 2 - Port Congestion

Representing flows

A flow table comprises multiple flow rules, which are match-action tuples. By creating match and action classes and establishing an object-type relation between them, we can capture the concept of flow rules. Flow-table class captures the flow rules (matches and associated actions). A packet can match multiple flow rules with different actions. Such flows are created by multiple network applications running on the controller. Though some actions can be performed serially, there will be actions whose occurrence is disjoint from other actions. Forwarding and filtering actions are examples of disjoint actions. The current version of ontology identifies such minor inconsistencies in flow rules.

Table 5.2 summaries object properties

Object property	Domain	Range	Inverse	Type
hasLink	Port	Link	ToPort	-
hasFlowTable	Datapath	FlowTable	-	-
hasMatch	FLowTable	Match	-	-
hasAction	Match	Action or Port	hasFlow	-
hasPort	Datapath	Port	PortOf	-
UsesLink	Match	Link	CarriesFlow	-
PortOf	Port	Datapath	hasPort	Functional
hasDP	Link	Datapath	DPhasLink	-
hasFlow	Port	Match	hasAction	-
hasHost	Datapath	Host	-	-
hasConnectionTo	Datapath	Datapath	-	Transitive
CarriesFlow	Link	Match	UsesLink	-
ToPort	Link	Port	hasLink	-
DPhasLink	Datapath	Link	hasDP	-
FTBelongstoDP	Datapath	FlowTable	-	-
hasNeighbour	Datapath	Datapath	-	-

Table 5.2: Object properties

5.4 Communication

Social ability is an essential aspect of multi-agent or distributed systems. Distributed systems, as briefly discussed in chapter 3, use one of the two modes of interaction (procedural or messages), and a middleware usually manages communication in distributed systems. In *Procedural* interaction, a component's methods are invoked by an external component using the component's global ID. Remote Procedural Calls (RPCs) access other component's exposed procedures via a stub. Similarly, in Remote Method Invocation (RMI), an object invokes methods of another object. Both components and objects are not autonomous, allowing external entities to access their methods and modify their internal state. In contrast, agents are autonomous and control decision-making, including deciding whether to act. Agents interact

by exchanging messages and not by remotely invoking other agents' methods, thus maintaining their autonomous nature. Agents exchange messages to request, query, inform and negotiate. This section details the proposed agent-based controller's communication framework starting with a brief overview of existing agent communication protocols and communication patterns in distributed systems. Agent communication patterns in the proposed agent system follow this discussion. Communication patterns are discussed using five common network scenarios.

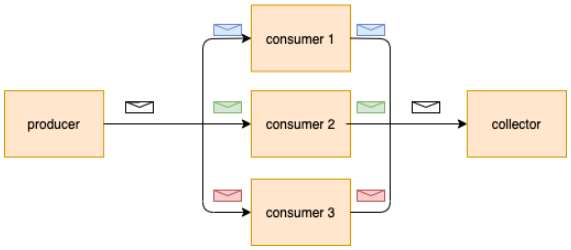
5.4.1 Communication in distributed systems

A communication pattern defines how messages are exchanged between two nodes via a message communication channel. Common communications patterns [153] are:

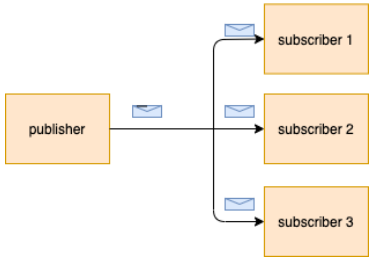
1. Pair messaging pattern has exactly two endpoints.
2. Client-Server messaging pattern has clients requesting data and a server attending to those requests by providing requested data. Also known as the Request-response pattern.
3. Push-Pull messaging pattern has producers pushing tasks on the consumers. Consumers sometimes push the results forward towards a collector, aggregating all results.
4. Publish-Subscribe messaging pattern has publisher publishing messages, and all the subscribers subscribed to this message receive and process the message

The push-Pull mechanism is different from the Publish-Subscribe mechanism. Smaller tasks are pushed on the workers /consumers in the push-pull mechanism. In the case of the publish-subscribe pattern, all subscribers receive the same message from the publisher [83]. Figure 5.15 illustrates the communication patterns listed above.

(a) Client-server messaging pattern



(b) Push-pull messaging pattern



(c) Publish-subscribe messaging pattern

Figure 5.15: Common messaging patterns in distributed systems

5.4.2 FIPA for agent communication

Human speech actions such as *ask*, *propose* have influenced agent communication language [92] and are called *performatives* or *speech acts*. Performatives play a similar role as their counterpart in the physical world. Agent communication languages such as Knowledge Query and Manipulative Language (KQML) [101], and Foundation for Intelligent Physical Agents (FIPA) [102] use speech performatives that operate on top of communication setup as illustrated in Figure 5.16. From a software implementation perspective, each performative is implemented as an agent method [154], [155], [156].

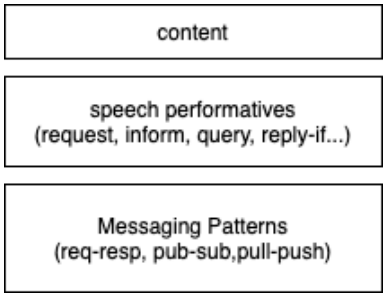


Figure 5.16: Communication acts operate on established communication patterns

FIPA [102] specifies standards for (a) agent communication, (b) agent management (c) agent message transport stages of agent interaction. Agent communication standard concerns itself with (a) interaction protocols, (b) communication acts and

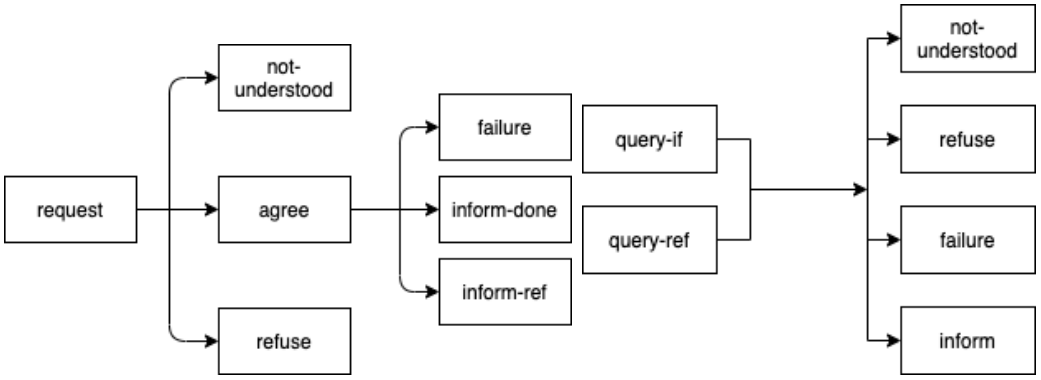
(c) content languages. Agent interaction protocols use communication acts (speech acts) for interaction. There are twenty-two communication acts agents use. A subset of speech acts used by agents in the proposed agent-based controller architecture are:

1. *inform* to convey information to an agent
2. *request* to request a receiver agent to perform some action
3. *failure* to convey to the requester agent of failure to execute an expected action
4. *not-understood* receiver agents use this performative to convey to the requester agent that a received message was not understood

A comprehensive list of all communicative acts proposed by FIPA standard is listed in [157]. Interaction Protocols describe ten interaction protocols between agents. FIPA request interaction and FIPA query interaction protocols are widely used interaction protocols. Various other FIPA interaction protocols are found in [158]. In FIPA request interaction protocol, an agent requests another agent to perform some action [159]. The receiving agent *agrees* and performs the actions and responds with the corresponding outcome using one of *failure*, *inform-done* or *inform-ref* acts. The agent can also *refuse* to act. This information is conveyed to the requester using *refuse* action. FIPA request interaction protocol is shown in Figure 5.17a. In FIPA query interaction protocol, an agent requests a receiving agent to perform an *inform* action [160]. The receiving agent uses one of the four communicative acts - *not-understood*, *refuse*, *failure*, *inform* to reply to the querying agent. FIPA query interaction protocol is shown in Figure 5.17b.

FIPA message standards defined are [161]. Some of the main fields in messages are :

1. *type of communication act* to capture performative
2. *participants in communication* to identify sender and receiver
3. *message content* to store content of message



(a) FIPA Request interaction protocol (b) FIPA Query interaction protocol

4. *control of conversation* to capture the FIPA interaction protocol in use.

5.5 Agent interactions

This section discusses agent communication patterns in common network scenarios ranging from simple monitoring functions to complex management functions. Agents exchange messages as per FIPA request protocol.

5.5.1 Discovering new links

TOagent periodically discovers links by requesting OFagent using a *request* message. OFagent in turn generates a LLDP packet and collects the responses, and relays the responses back to TOagent as a *inform* message. In the event of a port's state change to *up*, OFagent informs TOagent. TOagent updates KB and requests OFagent to initiate a complete link discovery process. This ensures correct and updated link information to knowledge base (KB) by KBagent. Agent interactions are shown in Figure 5.18.

The message exchanged during the process are listed in Table.5.3. The list of links in the message is a list of connected datapath and port and is of the form $\{(dp_1, pr_1) : (dp_2, pr_1), (dp_2, pr_2), (dp_3, pr_1)...\}$ where dp_i is a datapath ID, and pr_i is port number.

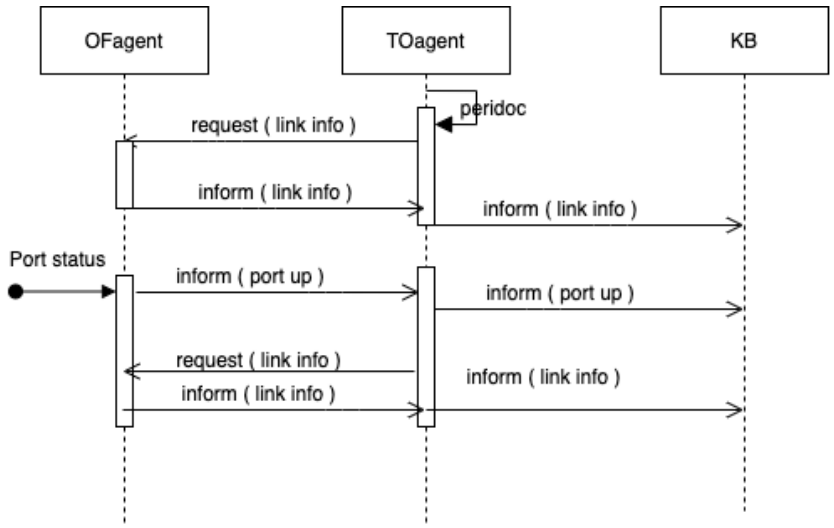


Figure 5.18: Discovering new links

Type	Sender	Receiver	Act	Content
new link	TOagent	OFagent	request	topo::
new link	OFagent	TOagent	inform	new_link::[list of links]
new link	TOagent	KBagent	inform	new_link::[list of links]
port up	OFagent	TOagent	inform	port_up::[sw,pr]
port up	TOagent	KBagent	inform	port_up::[sw,pr]
port up	TOagent	OFagent	request	topo::
port up	OFagent	TOagent	inform	new_link::[list of links]
port up	TOagent	KBagent	inform	new_link::[list of links]

Table 5.3: Messages exchanged while discovering new links

5.5.2 Provisioning

OFagent parses incoming Packet-In message from data plane and *requests* IPagent for a path to provision the flow on. IPagent uses a recent copy of topology to compute a path and *informs* OFagent and updates KB.

Agent interactions while provisioning a flow are shown in Figure 5.19, and corresponding messages exchanged are listed in Table 5.4

Flow_path information of form path::$flow$ :: $[(dp_a, pr_a), (dp_b, pr_b), ..., (dp_n, pr_n)]$ where (dp_i, pr_j) is (dpid, outport) tuple of intermediate datapath is sent to OFagent.

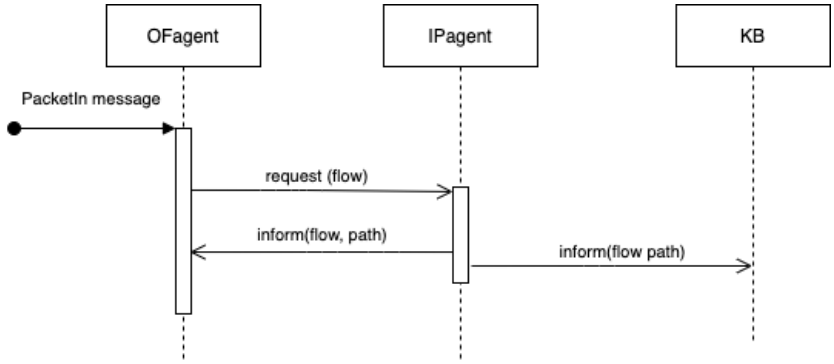


Figure 5.19: Provisioning a new flow

Type	Sender	Receiver	Act	Content
provision	OFagent	IPagent	request	path::flow
provision	IPagent	OFagent	inform	path::flow::flow_path
provision	IPagent	KBagent	inform	path::flow::flow_path

Table 5.4: Messages exchanged during flow provisioning

Flow is a tuple of form $(srcaddr, dstaddr, protocol)$ where srcaddr is the source address, dstaddr is the destination address, and protocol is application information.

5.5.3 Recovering from link failures

Two common network abnormalities previously discussed are a) link failure and b) congestion. Unlike network congestion, a link failure incident is registered when PDagent receives a port state change to *down* information from OFagent. PDagent updates the internal network state stored in the KB, as seen in Figure 6.11. KB responds with information on all affected flows. PDagent does not wait until flow-rule times out, rather proactively deletes flow rules from the datapath and triggers reprovisioning.

5.5.4 Addressing congestion

In case of congestion, there is no explicit notification from the data plane. PSagent updates the port's information. KB infers if a port is congested. In case a port is

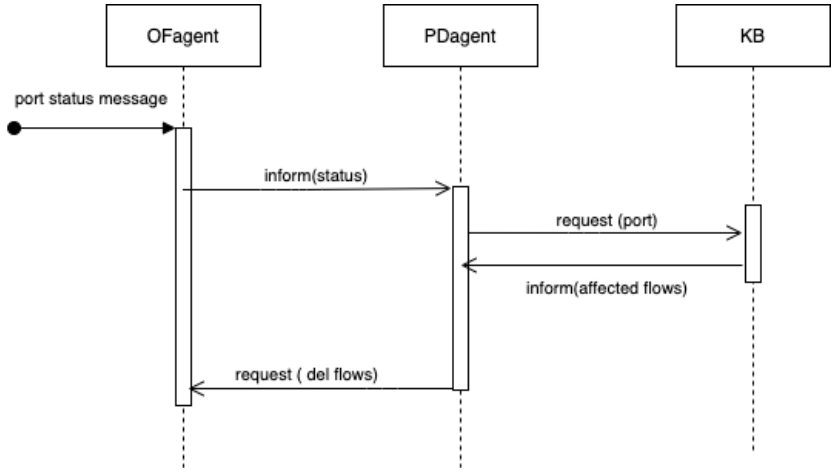


Figure 5.20: Handling port down event

Type	Sender	Receiver	Act	Content
port down	OFagent	PDagent	inform	port_down::(dp,pr)
port down	PDagent	KBagent	request	port_down::(dp,pr)
port down	KBagent	PDagent	inform	affected_flows::[list of flows]
port down	PDagent	OFagent	request	affected_flows::[list of flows]

Table 5.5: Messages exchanged during link failures

congested, the KBagent *requests* RRagent find alternate paths to provision affected flows. RRagent *requests* flow demands (fd_f) from OVagent and attempts to find alternate paths to configure affected flows. The RRagent *requests* IPagent to provision flows on new paths. While requesting IPagent to provision might occur as an extra step in the process, two reasons validate this step. First, each agent in an agent system performs a specific task. While an agent can perform multiple tasks, multiple agents cannot perform the same task. This is as per the agent roles identified in Chapter 4. Secondly, IPagent computes the end-to-end path, which is a combination of datapath and outgoing port, whereas RRagent only computes the path in terms of datapaths. Hence to configure flows on the new path, RRagent *requests* IPagent to provision. While IPagent configures new flow rules, old flow rules timeout after the idle-timeout period and FTagent purges KB of stale flow , path information by removing port, flow associations and flow table flow rule associations.

RRagent finds only those paths whose loads are within the constraints of the available link bandwidth after provisioning new flows. Figure 5.21 shows agent interactions, and Table 5.6 lists messages exchanged during the process.

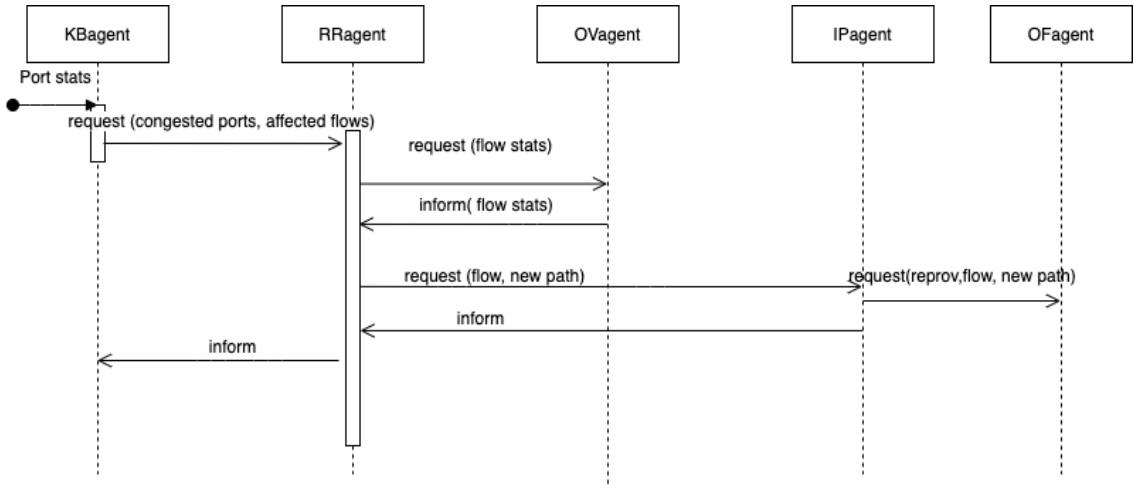


Figure 5.21: Agent interaction during link congestion

Type	Sender	Receiver	Act	Content
congestion	KBagent	RRagent	request	congestion::(dp,pr)
congestion	RRagent	OVagent	request	flow_stats::(dp,pr)
congestion	OVagent	RRagent	inform	flow_stats::[flows_stats]
congestion	RRagent	IPagent	request	reprov::[(flows::path)]
congestion	IPagent	OFagent	request	reprov::[(flows::path)]

Table 5.6: Messages exchanged during link failures

5.5.5 Maintaining fair resource allocation among TCP flows

TFagent uses pre-trained neural network model to control the *cwnd* of fast sender. The agent makes observations and, based on an action chosen, *requests* OVagent to install a temporary CE-bit enabled flow rule.

Figure 5.22 shows agent interactions, and Table.5.7 lists messages exchanged during the process.

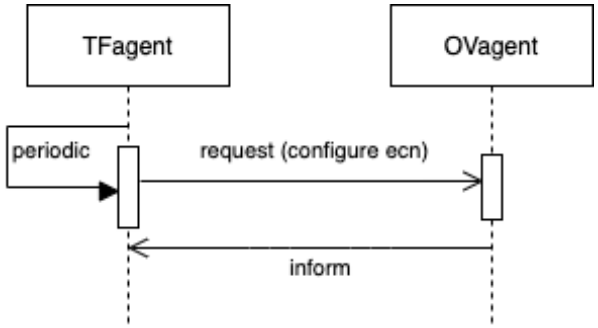


Figure 5.22: Agent interaction to ensure fairness

Type	Sender	Receiver	Act	Content
control cwnd	TFagent	OVagent	request	flow rule
control cwnd	OVagent	TFagent	inform	status

Table 5.7: Messages exchanged to ensure fairness

5.5.6 Gathering port counter information

PSagent periodically *requests* OFagent to query port stats. KB is updated with the information. TPagent uses this information to predict network traffic. Figure 5.23 shows agent interactions for gathering port stats and performing incremental traffic prediction on links, and Table.5.8 lists messages exchanged during the process.

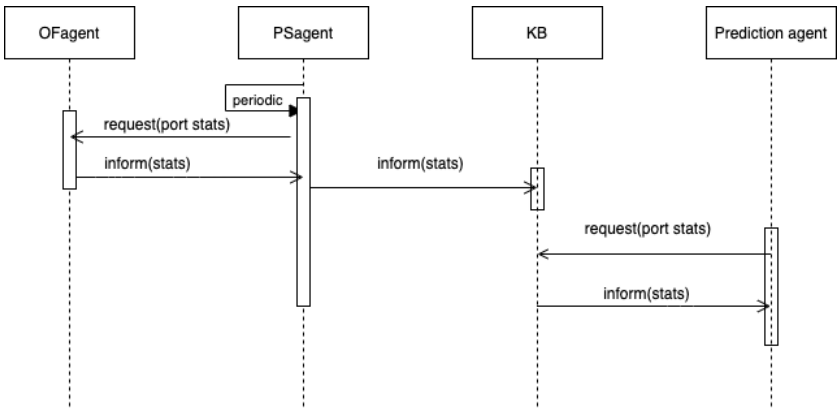


Figure 5.23: Agent interaction during monitoring port counters

Port counters received from OFagent are of form $\{dp_1 : \{pr_1 : val_1, pr_2 : val_2, ..pr_n : val_3\}\}$

Type	Sender	Receiver	Act	Content
port stats	PSagent	OFagent	request	port_stats::
port stats	OFagent	PSagent	inform	port_counters::[port counters]
port stats	PSagent	KBagent	inform	port_counter::[port counters]
port stats	KBagent	TPagent	inform	readings::[port counters]

Table 5.8: Messages exchanged during port statistics gathering process

5.6 Conclusion

This chapter introduced an ontology to formalise the information in the agent system. Various roles, restrictions and rules of ontology are established. Operation of KB is explained in terms of inferences made. KBagent operates on KB and enables other agents to communicate with KB. Next, interactions among the agents including message format and communication patterns were discussed. The agents exchange FIPA complaint messages for communication. In the next chapter, implementation and results are presented.

Chapter 6

Implementation

6.1 Introduction

This chapter evaluates the prototype build for the proposed agent-based SDN controller in two sections. The first section demonstrates the operations of three agents. The later section examines distributed agent controller's performance and makes a comparison against monolithic controller- Ryu [7] in various controller operational modes. The comparison is in terms of flow set-up duration and bandwidth conducted in network scenarios listed in chapter 5.

6.1.1 Prototype Setup

The proposed prototype is implemented in Python 3.9. Osbrain [162] is a Python-based multi-agent platform. Each agent is a system process, and multiple agents communicate using Inter-Process Communication IPC or TCP in multi-node environments. Agents use a broker-less messaging queue- ZeroMQ [83] to exchange messages with each other in one of the communication patterns - Publish-Subscribe, Push-Pull and Request-Response mechanisms discussed in Chapter 5. The Osbrain platform is generic and does not prescribe any specific agent architecture, hence suitable for building a proposed prototype.

OFagent is a multi-threaded agent that subscribes to both Osbrain's ZMQ message loop to exchange messages with other agents and to Asyncio [163] event loop to exchange messages with the data plane. The asyncio event loop and ZMQ messaging loop run on a dedicated threads. OFagent uses a lightweight python library PyOF [164] to parse OpenFlow packets. This prototype uses OpenFlow v.1.3.

6.1.2 Operation modes

The controller prototype operates in two modes as below:

1. Single node: Agents are co-located on the same physical node and communicate using IPC to operate as a single system while still exhibiting all the characteristics of an agent-based system.

2. Multiple nodes (distributed) - Agents are located on different physical devices and require an operational network set-up to communicate over TCP.

A controller in distributed set-up does not differ from a single-node set-up in functionality, as the agents are isolated in both operational modes and interact by exchanging messages. In comparison, the agents in single-node mode use either IPC or TCP for communication and agents in distributed mode communicate over TCP and require a fully functional communication network as a requirement. The tests were conducted for the topology shown in Figure 6.1 (unless mentioned otherwise) emulated in a well-known mininet [165] topology emulator. The topology is a two-tier network consisting of five switches (s1, s2, s3, s4, s5) and six hosts (h1, h2, h3, h4, h5). Links connecting adjacent neighbours, for example, the link connecting s1 and s4, is labelled as l14. The links are unidirectional; hence link l14 is not the same as l41. While the topology is simple, the experiments demonstrate the operation of the agent-based controller and agents.

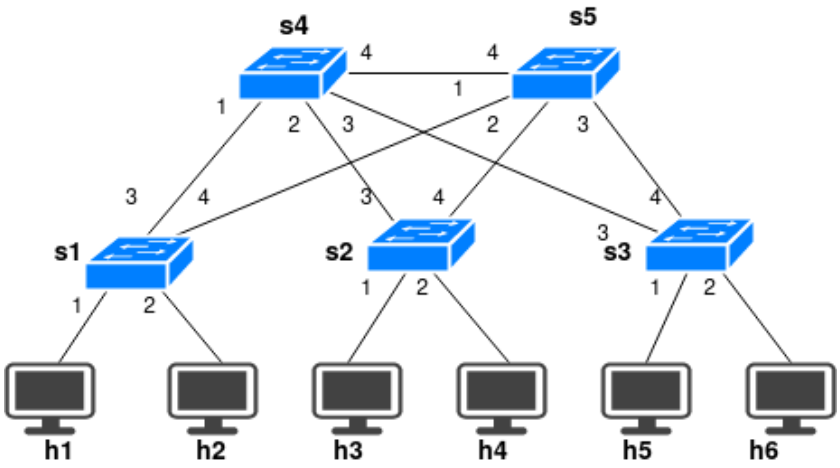


Figure 6.1: Emulated topology for testing operation of agent-based controller

6.2 Functional tests to verify agent operations

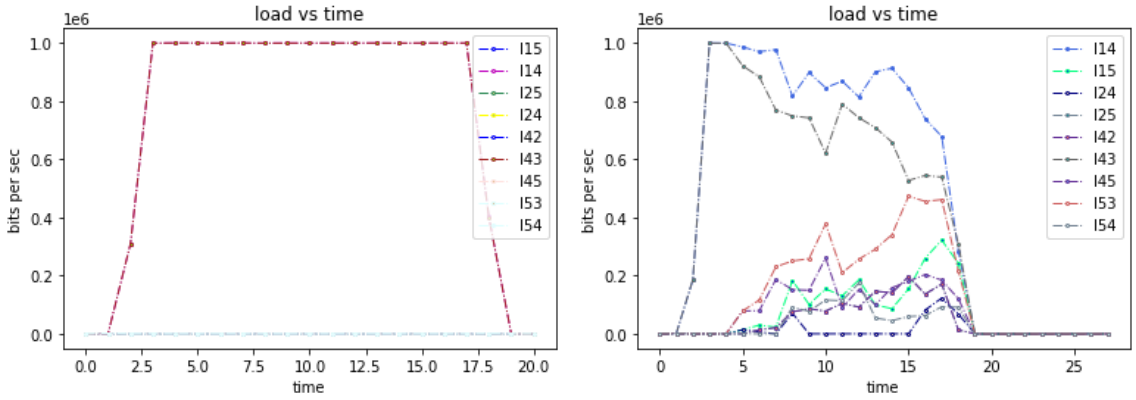
Three agents discussed in this section are the RRagent, the TFagent and TPagent. The RRagent is tasked with rerouting active flows to alleviate congestion by evenly distributing flows across possible paths. The TFagent adjusts the congestion window of fast senders to allocate bandwidth fairly across multiple flows, and the TPagent predicts link loads incrementally. This section provides the reader with results from functional testing of the for-mentioned agents. Experiments in this section were conducted on the single-node mode of the agent system as these agents are autonomous, and their functionality remains same across different operational modes.

6.2.1 Rerouting agent

This test verifies the operation of the RRagent. The RRagent receives congestion notification from KBagent and attempts to reroute a subset of active flows. For this experiment, each link's bandwidth is set to 1 Mbps. Dijkstra's shortest path algorithm (SPF) assigns flows to paths with the least cost. Link costs are updated periodically. The IPagent chooses a path based on the last received statistics at a given instance. For all flows initiated in parallel, originating from h1 towards h5, the IPagent chooses the same least-cost path, such as (s1-s4-s3), to configure flows. Such

choice results in fully utilising links l14 and l43 and underutilising other possible paths such as (s1-s5-s3). This behaviour is seen in Figure 6.2a.

Rerouting a subset of active flows to an alternate path (s1-s5-s3) reduces utilisation on links l14 and l43, while links l15 and 53 are better utilised, as seen in Figure 6.2b.



(a) Link load prior to rerouting

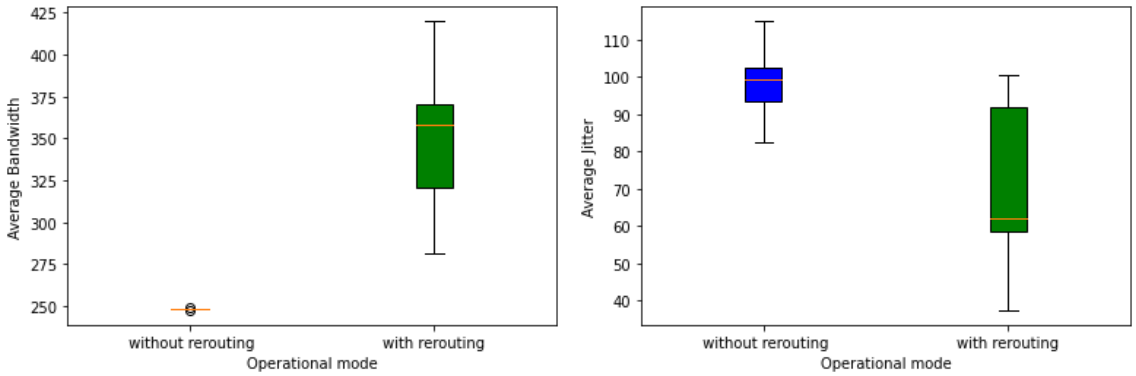
(b) Link load after rerouting

In Test 2, we verify the effect of rerouting on the jitter and bandwidth. All links operate at 1 Mpbs bandwidth. The link connecting s3 and h5 is set to operate at 2Mbps. Each host h1, h2, h3 and h4 generates UDP traffic for 60 secs towards h5 and averages are computed across hosts h1, h2, h3 and h4. Network topology information is collected every 20 secs.

Using SPF IPagent identifies the same path as the best path for all the active flows. Provisioning all flows on a single path leads to higher average jitter and lower average transfer rate.

On the other hand, rerouting a subset of active flows on alternate paths leads to higher average transfer rates resulting from higher average bandwidth and lower average jitter across hosts h1, h2, h3, and h4. as shown in Figure 6.3a and 6.3b respectively.

Discussion The maximum total number of possible solutions computed by the agent is p^f where p is the number of paths for f number of flows. A high number of solutions are generated when a minimum number of branches are pruned during the backtracking process, which implies a high number of successful flow assignments



(a) Avg. bandwidth comparison with and without rerouting

(b) Avg. jitter comparison with and without rerouting

Figure 6.3: Measuring bandwidth allocated and jitter experienced with or without rerouting of active flows.

satisfy the constraints. This is possible only when many paths have sufficient capacity to accommodate most of the flows. In this case, we can conclude that the paths are not highly congested- thus making the rerouting process redundant. This behaviour was observed when the threshold value was set to 70%.

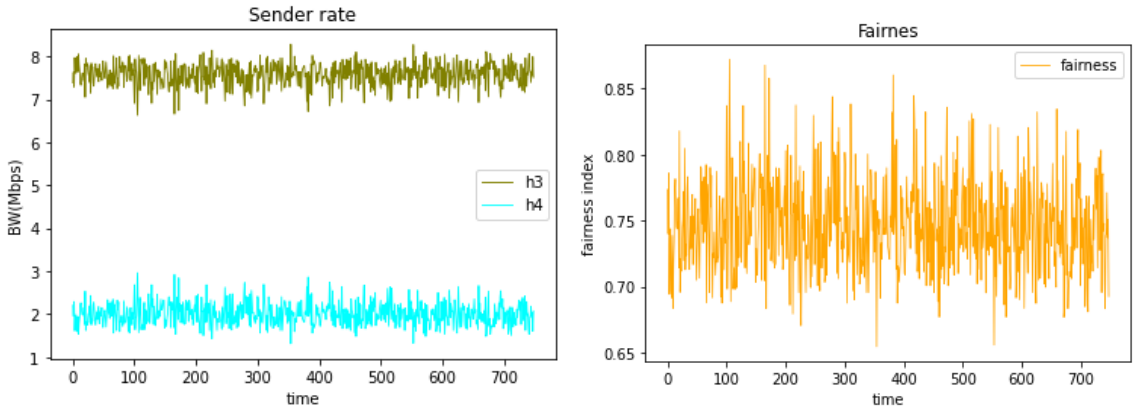
In other words, the agent performs worst when there is less congestion in the network. Hence the congestion threshold has been set to 80%.

In case where links are fully congested, the RRagent does not yield any solution as no alternative paths can be found that satisfy the constraints. Also, it was observed that longer polling intervals reduce the benefits of rerouting, because the rerouting process occurs only after a polling event. Longer polling intervals let the link stay congested for a long.

6.2.2 TCP fairness agent

This test verifies the TCP agent's operation in increasing fairness across TCP flows, as mentioned in chapter 4. The agent learns to maintain a high fairness index while ensuring not to throttle throughput to a drastically low rate, resulting in the under-utilisation of available bandwidth. In other words, the agent is punished if the congestion window is reduced when the action might result in low throughput. All

links operate at 10Mbps for topology in Figure 6.1 for this test. All links experience a delay of 10ms, while the link connecting s2 to h4 experiences a delay of 30ms. The rationale behind this configuration is that increase in RTT results in TCP increasing host h4's *cwnd* at a slower rate than host h3 resulting in a lower sending rate as seen in Figure 6.4a. This results in an average Fairness index as seen in Figure 6.4b.



(a) Throughput of host h3 , h4

(b) Jain's fairness index

Figure 6.4: Unfair sending rates of hosts h3 and h4

TFagent learns to reduce the congestion window of a fast sender by falsely indicating congestion using the CE flag in the IP header. The agent uses a pre-trained model to choose actions based on current observation. As mentioned in chapter 4, observation is a list of current sending rates of sources. The agent has three actions to choose from - **0** not to act, **1** to reduce the sending rate of a slow sender, and **2** to reduce the sending rate of a fast sender. Though the agent operates autonomously, the agent interacts with the OVagent to install flow rules on the terminal datapath s3.

As mentioned earlier, the agent uses a pre-trained model to select an action. An Open AI gym [166] environment was created to train the agent and build a model for the agent to use. Prior to this work [167], [168] developed gym environments for mininet. [167] proposed Iroko framework, [168] builds a gym environment to train the agent to select primary MPLS up-link over Internet up-link.

Open AI Gym [166] provides a standard interface for researchers and developers to develop and test their Reinforcement learning agents, allowing consistent testing and bench-marking of various algorithms. Though the environment for data networks is unavailable, OpenAI gym provides guidelines for building custom environments. A custom environment built to train the TFagent interacts with topology emulated in mininet, as shown in Figure 6.5.

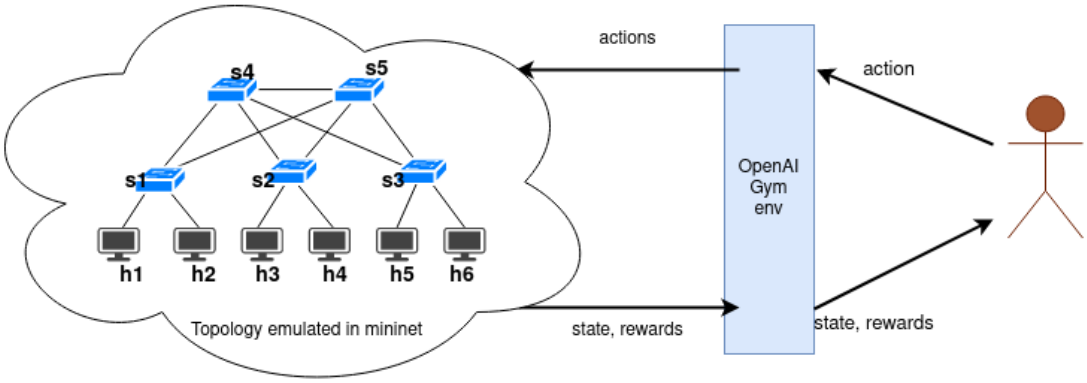


Figure 6.5: Mininet-OpenAI gym env

A custom gym environment maps domain-specific actions, such as installing flow rules to discrete actions of the DQN agent. For example, action **2** is mapped with the OVagent installing a temporary flow rule in Listing 1 to flag congestion to a fast sender.

```
ovs-ofctl add-flow s3 -O OpenFlow13 priority=20,ipv4,hard_timeout=1,nw_src=<nw_src>,
nw_dst=<nw_dst>,action=mod_nw_ecn:3,output:<out_port>
```

Listing 1: Flow rule to enable CE flag

where *nw_src* is the fast sender's IP address. Similarly, the environment calculates and rewards the agent based on observations of hosts h3 and h4 sending rates.

A custom Open-AI gym environment has

1. *initialise method* to initialise the environment.

2. *step method* to capture the 'observe -act- reward' cycle.

3. *reset method* to reset the environment at the end of the execution episode.

4. *close method* to close the environment.

An agent performs possible action from the *action_space*. Action_spaces are either:

1. *Discrete* to store discrete actions

2. *Box* store continuous actions

The agent observes the consequence of choosing an action in the form of environmental change. An observation is a snapshot of the environment. *Observation_space* is a collection of all such snapshots. Observation_spaces can themselves be of various formats as:

1. *Box* to store continuous observations

2. *Discrete* to store discrete observations

3. *Tuple* to store observations with multiple values

4. *Dictionary* similar to Tuple, but stores different features of observations using key: value pairs.

The TFagent is trained for 750 episodes. Since the agent is a reinforcement learning agent, the agent starts by exploring the environment. The exploration rate is set to 1.0 and decays at a rate of 0.95. Each episode runs until a terminal state is encountered. The terminal state is that nodes h3 and h4 are no longer connected to h5. Hosts h3 and h4 transmit data for 300 secs.

The rewards gained by the agent are shown in Figure 6.6.

Higher rewards also translate into better sending rates for h4 and better fairness index, as shown in Figure 6.7a and Figure 6.7b, respectively.

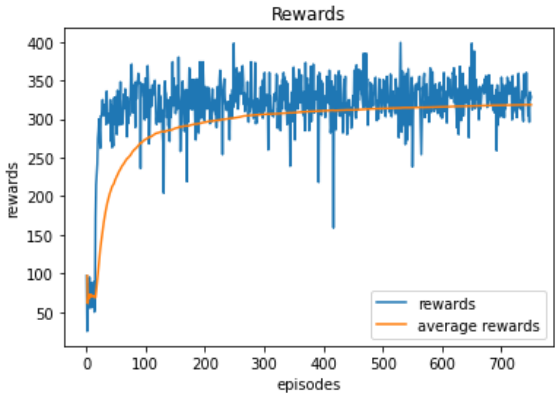
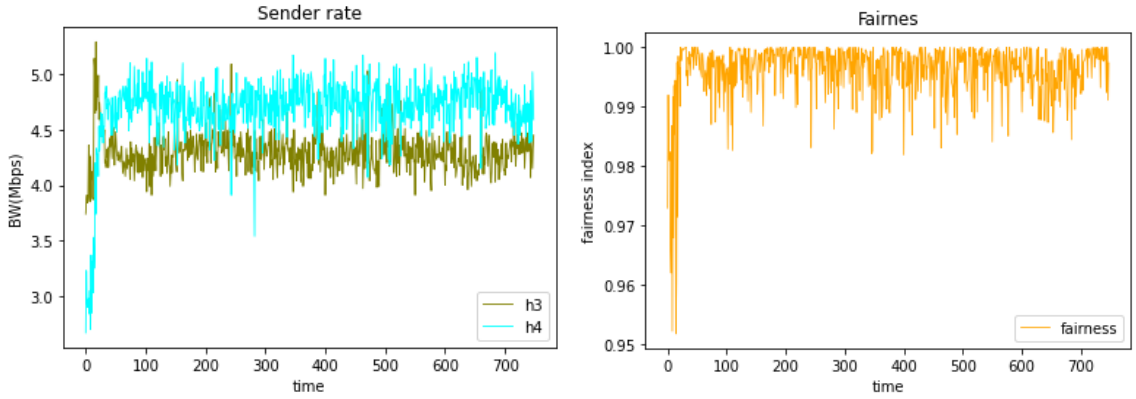


Figure 6.6: Rewards received by the TFagent



(a) Throughput of host h3 , h4

(b) Jain's fairness index

Figure 6.7: Fair sending rates of hosts h3 and h4 achieved by TCP agent

Discussion During initial episodes, the agent explores the action space based on an epsilon-greedy algorithm. The agent explores all actions with random chance and collects rewards for acting on that specific observation. As the episodes progress, the agent explores less and exploits actions with maximum rewards, thus proactively working towards gaining more rewards. Working *proactively* does not necessarily imply consistently reducing the congestion window. For example, consider an instance when h3 is transmitting at 4Mbps, and h4 is transmitting at 2Mbps on a 10Mbps link. The current sending rates remain unchanged by choosing action 0 (no action). A second choice (action 1) is to reduce the sending rate of the slow sender (i.e. h4). This action reduces the sending rate of h4 from 2Mbps to 1 Mbps. Third choice (action 2) is that the agent can reduce the fast sender's congestion window (i.e.

h3). Third action reduces the sending rate of h3 to 2Mbps. By choosing action 2 a Fairness index is 1 is achieved but at the cost of utilising 4Mbps of 10Mbps available bandwidth. Such action selection is undesirable, and the agent does not reap any rewards. Similarly, by choosing action 1, neither a higher fairness index nor better throughput is achieved. Thus a suitable action for this observation is action 0- to not act.

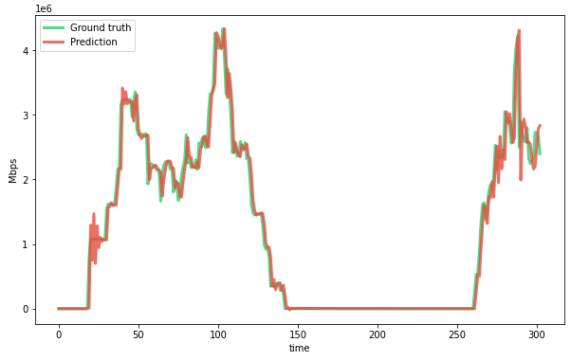
Also, in order to not cause a prolonged decrease in the fast sender's sending rate, a hard timeout of 1 sec has been set for the flow rule flagging congestion. This rule thus causes a temporary slow down of the fast sender by reducing the congestion window by half. Once the rule expires after 1 sec, the fast sender resumes additively, increasing the congestion window.

Since only two flows were considered for this test, the agent learnt in relatively fewer episodes. The scale of the experiment must be increased to test the agent's behaviour for a more significant number of flows. Further work is also required on reducing the neural network's training time. Training the neural network consumes time, and the observations made no longer reflect the effect of the action performed if the training time is greater than the observation interval. This is a general experimental observation made during the build stage.

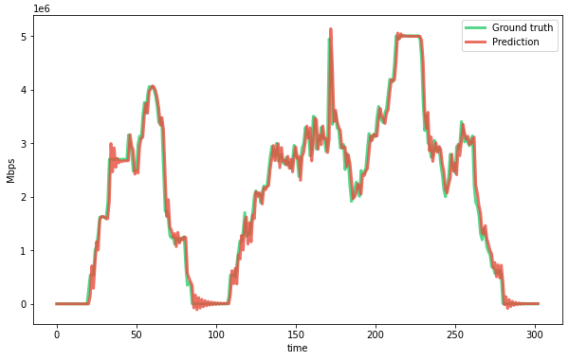
6.2.2.1 Traffic prediction agent

The TPagent receives port status information from KBagent. The agent predicts traffic on a link for a one-step ahead time stamp. The topology shown in Figure 6.1 has been used. All links operate at a bandwidth of 10Mbps. Hosts h1, h2, h3, and h4 generate UDP traffic for random duration between 1 to 3secs towards h5 and h6 in ON_OFF fashion before sleeping for a few seconds. This process is repeated for 60 mins, and corresponding true values are fed to TPagent one at a time to make a one-step-ahead prediction. Parameters used to train ARMA model are p = 3, d = 0, q = 1, initial_intercept =5 , SGD=0.01, learning_rate = 0.3). The agent calculates the Mean Absolute Error MAE of the predicted value in the next timestamp after

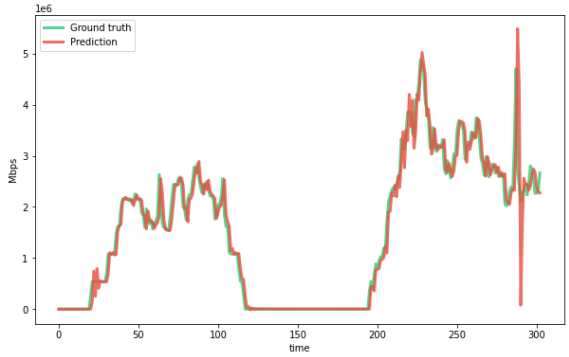
receiving the true value. TPagent uses Python library River [169] for incremental learning.



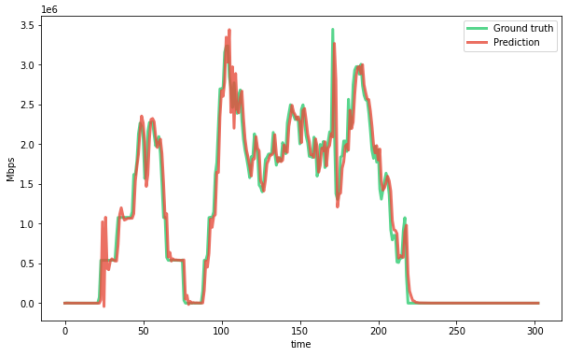
(a) Traffic on link l14



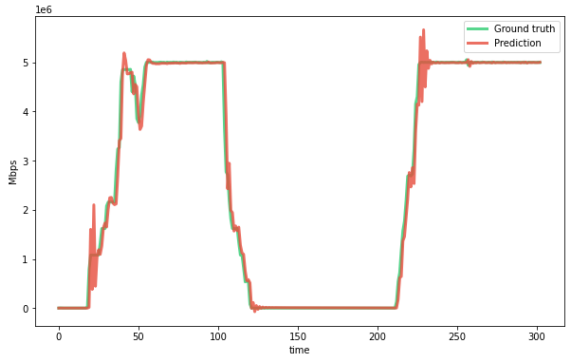
(b) Traffic on link l15



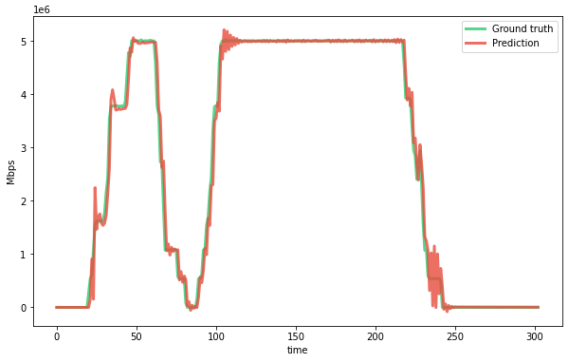
(c) Traffic on link l24



(d) Traffic on link l25



(e) Traffic on link l43



(f) Traffic on link l53

Figure 6.8: Predicting traffic on various links

Discussion There were two possible ways to train the model- by considering all links as features for a single model and training one model with traffic from all links or training a unique model for each link. Since TPagent learns incrementally and does not consume resources, second approach was preferred where each link was mapped to a unique model. Such a mapping accommodates network changes, such

as the addition or removal of links or changes in traffic on a specific link does not require retraining a single model with a changed number of links. To add new links, a new model is created.

The parameters for the current models were chosen based on the model with the least MAE. Unlike batch training, cross-validation cannot be performed in incremental learning as the agent does not store datasets that can be split into training and testing datasets. Finally, in batch learning, parameters p,q is chosen using ACF and PACF. Since the entire data set is not available , parameters p,q have been chosen based on experimentation. Values of p,q that yielded the lowest MAE have been chosen. Traffic predictions by TPagent for links l14, l15, l24, l25 and l43 and l53 are shown in Figure 6.8.

6.3 Functional and Performance evaluation

In this section, the controller's response time for setting up a single flow on a multi-hop network, for setting up multi-flows in multi-hop topology and recovery from link failures in single node operational mode and distributed mode is measured and compared against response times of Python-based Ryu [7] SDN controller. The agent system is set to operate on a single Linux device in the single node operational mode. The agent system is deployed on Raspberry Pi units for a distributed set-up. A subset of controller agents is used for the evaluations. For the distributed mode of operation, OFagent and KBagent run on a Linux machine, as handling the data plane connections and maintaining knowledge graphs are resource-intense activities. TOagent, IPagent, PSagent and FTagent agent each run on individual Raspberry Pis. All the agents in the agent system communicate over TCP on wireless channels. Distributed set-up using Raspberry Pis is shown in Figure 6.9.

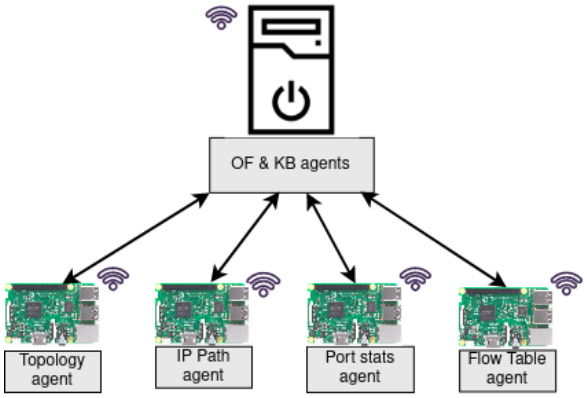


Figure 6.9: Agents on Raspberry Pi

6.3.1 Provisioning

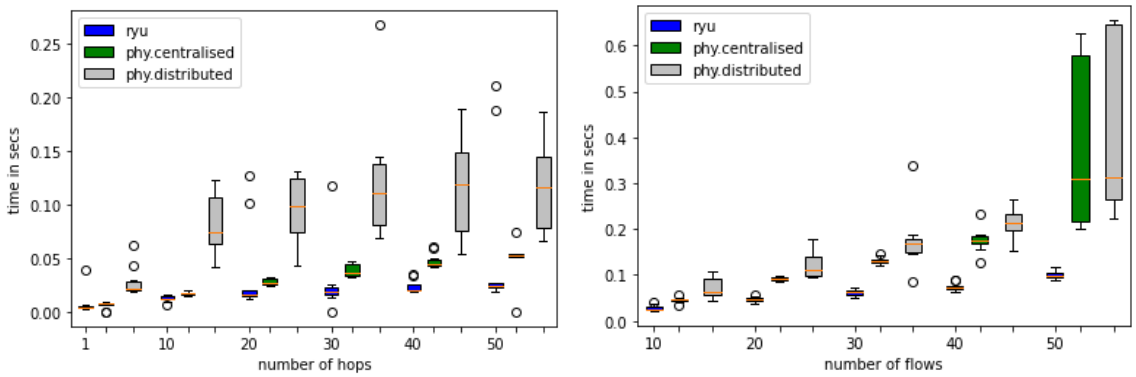
This section provides results of provisioning multiple flows and a single flow on a multi-hop path. Agent-based controller's response time calculated for comparison purposes.

Setting up a flow

A linear topology of fifty switches is used to compute response time for provisioning a single flow on paths of varying hop lengths. h1 connected to switch s1 is the source and progressively sends packets to h2, h3, ..., h25, h26,..., h50 connected to switches s2, s3,...,s25, s26,...s50. Flow set-up duration is calculated as the time elapsed between an arriving Packet-In message and the final outgoing Flow-Mod message from the controller. The longer the hop count, the higher the flow set-up time. Ten such *runs* were conducted to measure flow set-up times for path lengths from 1 to 50. The test was repeated on both agent-based controller operational modes and compared against the Ryu controller's set-up duration. The average flow set-up time for x hops across all runs is calculated using time stamps obtained from Wireshark [170]. Results are shown in Figure 6.10a.

Configuring multiple TCP flows

The topology shown in Figure 6.1 is used to compute flow set-up time for multiple flows. h5, h6 are destination nodes and h1, h2 generate TCP traffic towards the h5, h6. A *run* consists of generating an increasing number of flows [5, 10, 15,...,45, 50] and calculating flow set-up times for x number of flows. An average flow rule set-up time and reverse flow rule set-up time are computed as set-up time for x number of flows. As with the earlier test, the test was performed on both operational modes of the agent controller and compared against the Ryu controller's set-up time. Each run's packet's time stamps were extracted using Wireshark. The results of this test are shown in Figure 6.10b.



(a) flow set up time over multiple hops (b) flow set up time for multiple flows

Figure 6.10: comparison of flow set-up duration

6.3.2 Recovering from link failures

This test aims to evaluate the duration the agent system takes to react and recover from link failures in terms of the time taken by the controller to re-provision an affected flow. The topology shown in Figure 6.1 is used for the test. For every run, a link is shut down ten times and the time taken by the controller to configure a new flow rule to reroute the affected flow is computed. Response time is calculated as time lapses between the incoming Port-down message and the final outgoing flow-mod message. Timestamps for these packets were obtained from Wireshark.

As with previous experiments, agent controller response times have been compared with the Ryu controller's response time. It has to be noted that the agent system reroutes affected flows proactively. The controller reroutes the flows upon receiving the Port-down message and does not wait for the Packet-In message. Figure 6.11 captures the response time for this test.

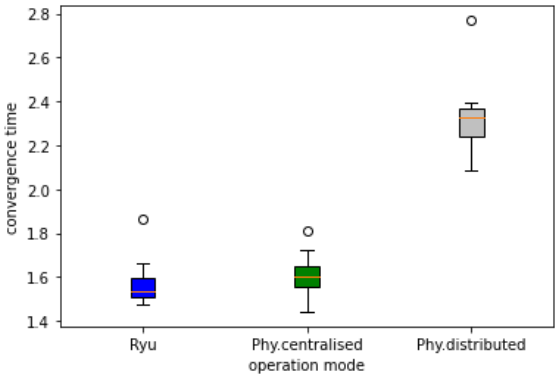


Figure 6.11: Handling port down event

Discussion Figures Figure 6.10a, Figure 6.10b, and Figure 6.11 show that the physically distributed set-up performs the worst, while the agent system operating on a single node performs relatively better. This is still promising as no optimisation techniques have yet been used to improve the system performance. Using multi-threading and other IO techniques can improve the performance of the system. However, this has not been the focus of this thesis. The system's performance in distributed operation mode depends on multiple factors, such as device capabilities, underlying communication hardware, and how libraries have been modified to operate on these devices.

6.4 Conclusion

This chapter presents evaluations and results of the proposed agent-based SDN controller. Complex agents' behaviours are evaluated individually, and the performance of the agent system as a whole is evaluated in common network scenarios. Tests

have been conducted for single and distributed modes of operation of the controller. It is observed that distributed set-up has a high flow-setup duration, whereas are single operation mode has relatively comparable results with Ryu. The next chapter concludes the thesis and discusses future work.

Chapter 7

Conclusion

This thesis presented a modular SDN control plane as an alternative to existing monolithic control planes to isolate components, enable flexible resource allocation. It also diversifies the SBI plane to develop multiple SBI to interact with the data plane.

This thesis proposed breaking the controller into a group of agents, by loosely adopting the MaSE methodology, to identify system goals. Agent roles were created to achieve system goals, and various agents performed various tasks to fulfil a network function.

Not all agents in the agent system are simple agents. Specifically, TOagent, IPagent, PSagent and FTagent are reactive agents. Advanced agents such as RRagent reason under constraints, whereas TFagent agent learns to reduce the *cwnd* of the fast sender. TPagent incrementally learns to predict network traffic.

The agents, while autonomous, interact amongst themselves to exchange information. An OWL-based Knowledge base is also presented. Reasoners like HermiT make inferences on the knowledge base to deduce new facts such as congestion, flows configured on ports, link down events.

The performance of the agent system is evaluated at the agent and system levels. At the system level, the proposed agent-based controller's performance while

operating on a single node is comparable to the monolithic Ryu [7] SDN controller. The controller, though, experienced latency in a physically distributed setup.

7.1 Further Work

The proposed agent system MASDN is modular, autonomous and social. Synchronisation and latency are two issues that affect distributed systems. These issues also affect the proposed agent system. Currently, the system uses a global timer to synchronise agents. More advanced techniques must be investigated and applied to attain global synchronisation while reducing latency.

As mentioned in chapter 3 and chapter 1, this thesis does not provide a framework for social decision-making. Interactions in the current prototype are limited to information exchange, and agents do not use advanced interaction protocols such as negotiations, bargaining and conflict resolutions to arrive at social decisions. Social decisions are those decisions taken by the agent system as a whole. In the context of SDN, social decisions are the prospective mechanism to achieve application composition. Ensuring OVagent and OFagent do not install conflicting flow rules in the datapath, or agent precedence over other agents. This can only be answered by employing cooperative decision-making ability in the agents leading to strong agency amongst agents. Achieving such strong agency among agents is a possible direction for future work in this area.

For experimentation purposes, it was assumed that traffic sources are well-behaved and transmit traffic at constant bit rate to verify the behaviour of RRagent and TFagent and TPagent and on-off nature of real network was recreated by switching traffic sources ON and OFF. Real-time network traffic is self-similar [171], [172] with periods of traffic bursts. Observing and updating the functionality of these agents based on realistic network traffic is an interesting future work. Such future upgrades also demonstrate the benefits of using an agent based controller, where the agents behaviour is constantly upgraded without impacting the overall system behaviour.

Improving the prototype's performance in terms of latency and number of active connections also requires investigation. While the prototype successfully demonstrates the operation of a modular SDN controller, OFagent's scalability to increase the number of datapath connections, improving efficiency of inter-agent communication are few of the system-level enhancements worth investigation. At the agent level, reducing the time taken to train TFagent, enhancing TFagent's ability to handle TCP flows and ensure fairness, and reducing the time taken by KBagent to make inferences need investigation is also a fruitful future investigation area.

The proposed agent-based architecture is the first of its kind. Though modular SDN controller designs have been presented in the past, at the time of writing, no working prototypes were described. With proposed future work and performance enhancements, the MASDN controller is a promising step towards building autonomous SDN controllers.

7.2 Conclusion

This thesis is a step towards an intelligent and autonomous SDN control plane. Firstly, the modular agent system is cloud-ready and edge-ready. It is possible to wrap agents into containers (for example, as dockers) and deploy them on the cloud. Container management systems such as Kubernetes can manage the containers. Flexible resource allocation and self-starting are some of the benefits. Similarly, as demonstrated in chapter 6, agents can also be deployed on end devices such as Raspberry Pis and Internet of Things devices. The proposed agent system is scalable. New agents for performing other network activities such as IP routing, VLAN forwarding, handling MPLS packets, load balancing, and firewalls can join the agent system without any change to the existing system.

Acronyms

ACF Auto-Correlation Function. 58, 61, 125

ACL Access Control lists. 13

AR Auto Regression. 58, 61

ARIMA Auto Regression Integrated Moving Average. 58

ARMA Auto Regression Moving Average. 58, 61, 123

ARP Address Resolution Protocol. 94

ASIC Application-Specific Integrated Circuit. 12

BGP Border Gateway Protocol. 12

CAM Content-Addressable Memory. 12

CNN Convolutional Neural network. 59

DL Descriptive Logic. 89

DQN Deep Q-Network. 120

FE Forwarding Element. 15

FIPA Foundation for Intelligent Physical Agents. 105–107, 113

FOL First Order Logic. 84, 89

FPGA Field Programmable Gate Array. 15

FTagent FlowTable agent. 57, 110, 125, 130

GRU Gated Recurrent Unit. 58, 59

IGMP Internet Group Management Protocol. 12

IP Internet Protocol. 80, 81, 93, 94, 119

IPagent IPPath agent. 54, 57, 63, 108, 110, 116, 117, 125, 130

IPC Inter-Process Communication. 114, 115

KB KnowledgeBase. 85, 107–109, 112, 113

KBagent KnowledgeBase agent. 63, 107, 110, 113, 116, 123, 125, 132

LLDP Link Local Discovery Protocol. 12, 51, 55, 107

LSTM Long Short Term Memory. 58, 59

MA Moving Average. 58, 61

MAC Media Access Control. 93, 94

MAE Mean Absolute Error. 58, 61, 62, 123, 125

MAS Multi-agent System. 35

MASDN Multi-agent Software-defined network. 6, 7, 45, 49, 75, 131, 132

NBA NorthBound Application. 3, 4

NBI NorthBound Interface. 3

OF OpenFlow. 3

OFagent OpenFlow agent. 51, 52, 54–57, 63, 107–109, 112, 115, 125, 131, 132

OSPF Open Shortest Path First. 12

OVagent OVS-vsctl agent. 52, 110, 111, 119, 120, 131

OWL Web Ontology Language. 50, 84–86, 89

P4 Programming Protocol-Independent Packet Processors. 15

PACF Auto-Correlation Function. 58, 125

PDagent PortDown agent. 63, 109

PISA Protocol Independent Switch Architecture. 15

PSagent PortStats agent. 56, 59, 62, 109, 112, 125, 130

QoS Quality of Service. 13

RDF Resource Description Framework. 84

RL Reinforcement Learning. 75

RMSE Root Mean Squared Error. 58, 61

RRagent Reroute agent. 63–65, 68, 70, 83, 110, 111, 116, 118, 130, 131

RTT Round Trip Time. 72–75, 80

SBI SouthBound Interface. 3, 4, 6–8, 14, 130

SDN Software-Defined Network. 3

SGD Stochastic Gradient Descent. 60, 61, 79

SNMP Simple Network Management protocol. 58

SPF Shortest Path First. 117

STP Spanning Tree Protocol. 12

TCAM Tertiary Content-Addressable Memory. 12

TCP Transport Control Protocol. 47, 51, 63, 71, 73–75, 80, 81, 83, 93, 95, 114, 115, 118, 119

TFagent TCPFairness agent. xii, 63, 80–83, 111, 116, 119–122, 130–132

TOagent Topology agent. 55, 63, 107, 125, 130

TPagent TrafficPrediction agent. 61, 62, 83, 112, 116, 123–125, 130, 131

UDP User Datagram Protocol. 93, 95

Bibliography

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, and J. Rexford, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008, ISSN: 07380658.

[2] J. Ingeno, "Architecting modern applications," in *Software architect's handbook: Become a successful software architect by implementing effective architecture concepts*. Packt, 2018.

[3] D. Comer and A. Rastegarnia, "Toward Disaggregating the SDN Control Plane," *IEEE Communications Magazine*, vol. 57, no. 10, pp. 70–75, 2019, ISSN: 15581896. DOI: 10.1109/MCOM.001.1900063. arXiv: 1902.00581.

[4] C. Manso, R. Vilalta, R. Casellas, R. Martinez, and R. Munoz, "Cloud-native SDN controller based on micro-services for transport networks," *Proceedings of the 2020 IEEE Conference on Network Softwarization: Bridging the Gap Between AI and Network Softwarization, NetSoft 2020*, pp. 365–367, 2020. DOI: 10.1109/NetSoft48620.2020.9165377.

[5] Q. P. Van, D. Verchere, H. Tran-Quang, and D. Zeghlache, "Container-based microservices SDN control plane for open disaggregated optical networks," *International Conference on Transparent Optical Networks*, vol. 2019-July, pp. 2019–2022, 2019, ISSN: 21627339. DOI: 10.1109/ICTON.2019.8840430.

[6] *pox*. [Online]. Available: https://github.com/noxrepo/pox.

[7] *Ryu SDN Controller*. [Online]. Available: https://osrg.github.io/ryu/.

[8] *NETCONF*. [Online]. Available: https://datatracker.ietf.org/wg/netconf/about/.

[9] *Simple network Management Protocol*. [Online]. Available: http://www.net-snmp.org.

[10] R. Elio, J. Hoover, I. Nikolaidis, M. Salavatipour, L. Stewart, and K. Wong, "About Computing Science Research Methodology,"

[11] M. Wooldridge, N. R. Jennings, and D. Kinny, "The gaia methodology for agent-oriented analysis and design," *Autonomous Agents and multi-agent systems*, vol. 3, no. 3, pp. 285–312, 2000.

[12] P. Giorgini, M. Kolp, J. Mylopoulos, and M. Pistore, "The tropos methodology," *Methodologies and software engineering for agent systems*, pp. 89–106, 2004.

[13] G. Caire, W. Coulier, F. Garijo, J. Gómez-Sanz, J. Pavón, P. Kearney, and P. Massonet, "The message methodology," in *Methodologies and Software Engineering for Agent Systems*, Springer, 2004, pp. 177–194.

[14] M. Winikoff and L. Padgham, "The prometheus methodology," in *Methodologies and software engineering for agent systems*, Springer, 2004, pp. 217–234.

[15] S. A. DeLoach, "The mase methodology," in *Methodologies and software engineering for agent systems*, Springer, 2004, pp. 107–125.

[16] S. A. DeLoach and J. C. Garcia-Ojeda, "O-mase: A customisable approach to designing and building complex, adaptive multi-agent systems," *International Journal of Agent-Oriented Software Engineering*, vol. 4, no. 3, pp. 244–280, 2010.

[17] P. Goransson, T. Culver, and C. Black, *Software Defined Networks*, 2nd ed. Morgan Kaufmann Publishers, 2017.

[18] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015, ISSN: 15582256. DOI: 10.1109/JPROC.2014.2371999. arXiv: 1406.0440.

[19] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 99–110, Aug. 2013, ISSN: 0146-4833. DOI: 10.1145/2534169.2486011. [Online]. Available: https://doi.org/10.1145/2534169.2486011.

[20] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[21] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch," ser. NSDI'15, Oakland, CA: USENIX Association, 2015, pp. 117–130, ISBN: 9781931971218.

[22] *Onf*. [Online]. Available: https://opennetworking.org/.

[23] ONF, "OpenFlow Switch Specification 1.4.0," *Current*, vol. 0, pp. 1–3205, 2013, ISSN: 09226389.

[24] *OF v 1.0*. [Online]. Available: https://opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf.

[25] *OF v 1.1*. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.1.0.pdf.

[26] *OF v 1.2*. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.2.pdf.

[27] *OF v 1.3*. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf.

[28] *OF v 1.4*. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.4.0.pdf.

[29] *OF v 1.5*. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[30] A. Doria, J. Hadi Salim, R. Haas, H. Khosravi, W. Wang, R. Gopal, and J. Halpern, *Forwarding and Control Element Separation*. [Online]. Available: https://datatracker.ietf.org/wg/forces/documents/.

[31] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," *HotSDN 2013 - Proceedings of the 2013 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, pp. 127–132, 2013. DOI: 10.1145/2491185.2491190.

[32] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM 2011 Conference*, 2011, pp. 254–265.

[33] M. Smith, M. Dvorkin, Y. Laribi, V. Pandey, P. Garg, and N. Weidenbacher, *OpFlex Control Protocol*, 2014. [Online]. Available: https://tools.ietf.org/html/draft-smith-opflex-00.

[34] Z. Latif, K. Sharif, F. Li, M. M. Karim, and Y. Wang, "A comprehensive survey of interface protocols for software defined networks," *arXiv*, pp. 1–30, 2019, ISSN: 23318422. arXiv: 1902.07913.

[35] O. Michel, R. Bifulco, G. Retvari, and S. Schmid, "The programmable data plane: Abstractions, architectures, algorithms, and applications," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–36, 2021.

[36] F. Bannour, S. Souihi, and A. Mellouk, "Distributed SDN Control: Survey, Taxonomy, and Challenges," *IEEE Communications Surveys and Tutorials*, vol. 20, no. 1, pp. 333–354, 2018, ISSN: 1553877X. DOI: 10.1109/COMST.2017.2782482.

[37] O. Blial, M. Ben Mamoun, and R. Benaini, "An Overview on SDN Architectures with Multiple Controllers," *Journal of Computer Networks and Communications*, vol. 2016, 2016, ISSN: 2090715X. DOI: 10.1155/2016/9396525.

[38] L. Zhu, M. Karim, K. Sharif, C. Xu, F. Li, X. Du, and M. Guizani, "SDN Controllers: A Comprehensive Analysis and Performance Evaluation Study," *ACM Computing Surveys*, vol. 53, no. 6, pp. 1–40, 2020, ISSN: 1557-7341. DOI: 10.1145/3421764.

[39] F. Benamrane, R. Benaini, *et al.*, "Performances of openflow-based software-defined networks: An overview," *Journal of Networks*, vol. 10, no. 6, p. 329, 2015.

[40] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 136–141, 2013.

[41] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008, ISSN: 0146-4833. DOI: 10.1145/1384609.1384625.

[42] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in Software-Defined networks," in *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*, San Jose, CA: USENIX Association, 2012. [Online]. Available: https://www.usenix.org/conference/hot-ice12/workshop-program/presentation/tootoonchian.

[43] *Faucet*. [Online]. Available: https://faucet.nz.

[44] *graphana*. [Online]. Available: https://grafana.com.

[45] D. Erickson, "The beacon openflow controller," ser. HotSDN '13, Hong Kong, China: Association for Computing Machinery, 2013, pp. 13–18, ISBN: 9781450321785. DOI: 10.1145/2491185.2491189. [Online]. Available: https://doi.org/10.1145/2491185.2491189.

[46] "FloodLight Controller," [Online]. Available: http://190.15.141.68/index.php/uta/floodlight-controller.

[47] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "Onos: Towards an open, distributed sdn os," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, New York, NY, USA: Association for Computing Machinery, 2014, pp. 1–6, ISBN: 9781450329897. DOI: 10.1145/2620728.2620744. [Online]. Available: https://doi.org/10.1145/2620728.2620744.

[48] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, S. Shenker, *et al.*, "Onix A Distributed Control Platform for Large.pdf," *USENIX Conference on Operating Systems Design and Implement*, vol. 10, pp. 1–14, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924968.

[49] A. Tootoonchian and Y. Ganjali, "HyperFlow: A distributed control plane for OpenFlow," *2010 Internet Network Management Workshop / Workshop on Research on Enterprise Networking, INM/WREN 2010*, 2010.

[50] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications," p. 19, 2012. DOI: 10.1145/2342441.2342446.

[51] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven sdn controller architecture," in *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, 2014, pp. 1–6. DOI: 10.1109/WoWMoM.2014.6918985.

[52] A. Dixi, F. Hao, S. Mukherjee, T. V. Lakshman, and R. R. Kompella, "ElastiCon: An elastic distributed SDN controller," *ANCS 2014 - 10th 2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 17–27, 2014. DOI: 10.1145/2658260.2658261.

[53] *Hazelcast*. [Online]. Available: https://hazelcast.com/.

[54] K. Phemius, M. Bouet, and J. Leguay, "DISCO : Distributed Multi-domain SDN Controllers," pp. 21–24,

[55] M. A. S. Santos, B. A. A. Nunes, K. Obraczka, and T. Turletti, "Decentralizing SDN ' s Control Plane," in *39th Annual IEEE Conference on Local Computer Networks*, IEEE, 2014, pp. 402–405, ISBN: 9781479937806.

[56] Y. Fu, J. Bi, K. Gao, Z. Chen, J. Wu, and B. Hao, "Orion: A hybrid hierarchical control plane of software-defined networking for large-scale networks," *Proceedings - International Conference on Network Protocols, ICNP*, no. October, pp. 569–576, 2014, ISSN: 10921648. DOI: 10.1109/ICNP.2014.91.

[57] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM Sigplan Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[58] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular sdn programming with pyretic," *Technical Reprot of USENIX*, vol. 30, 2013.

[59] A. Voellmy, H. Kim, and N. Feamster, "Procera: A language for high-level reactive network control," in *Proceedings of the first workshop on Hot topics in software defined networks*, 2012, pp. 43–48.

[60] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," *Acm sigplan notices*, vol. 47, no. 1, pp. 217–230, 2012.

[61] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi, "Tierless programming and reasoning for {software-defined} networks," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 519–531.

[62] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," *Acm sigplan notices*, vol. 49, no. 1, pp. 113–126, 2014.

[63] S. Layeghy, F. Pakzad, and M. Portmann, "SCOR: Software-defined Constrained Optimal Routing Platform for SDN," 2016. arXiv: 1607.03243. [Online]. Available: http://arxiv.org/abs/1607.03243.

[64] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory networking: An api for application control of sdns," *ACM SIGCOMM computer communication review*, vol. 43, no. 4, pp. 327–338, 2013.

[65] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," *Proceedings of NSDI 2010: 7th USENIX Symposium on Networked Systems Design and Implementation*, pp. 281–295, 2019.

[66] Y. C. Wang, Y. D. Lin, and G. Y. Chang, "SDN-based dynamic multipath forwarding for inter–data center networking," *International Journal of Communication Systems*, vol. 32, no. 1, pp. 2–4, 2019, ISSN: 10991131. DOI: 10.1002/dac.3843.

[67] J. Zhou, M. Tewariy, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat, "WCMP: Weighted cost multipathing for improved fairness in data centers," *Proceedings of the 9th European Conference on Computer Systems, EuroSys 2014*, 2014. DOI: 10.1145/2592798.2592803.

[68] S. M. Park, S. Ju, and J. Lee, "Efficient routing for traffic offloading in software-defined network," *Procedia Computer Science*, vol. 34, pp. 674–679, 2014, ISSN: 18770509. DOI: 10.1016/j.procs.2014.07.096. [Online]. Available: http://dx.doi.org/10.1016/j.procs.2014.07.096.

[69] M. A. Moyeen, F. Tang, D. Saha, and I. Haque, "SD-FAST: A Packet Rerouting Architecture in SDN," *15th International Conference on Network and Service Management, CNSM 2019*, 2019. DOI: 10.23919/CNSM46954.2019.9012703.

[70] R. Kanagevlu and K. M. M. Aung, "SDN controlled local re-routing to reduce congestion in cloud data center," *Proceedings - 2015 International Conference on Cloud Computing Research and Innovation, ICCCRI 2015*, pp. 80–88, 2016. DOI: 10.1109/ICCCRI.2015.27.

[71] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene, "Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks," *CoNEXT 2014 - Proceedings of the 2014 Conference on Emerging Networking Experiments and Technologies*, pp. 149–159, 2014. DOI: 10.1145/2674005.2674985.

[72] M. Shafiee and J. Ghaderi, "A Simple Congestion-Aware Algorithm for Load Balancing in Datacenter Networks," *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3670–3682, 2017, ISSN: 10636692. DOI: 10.1109/TNET.2017.2751251.

[73] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," *Proceedings - IEEE INFOCOM*, pp. 1629–1637, 2011, ISSN: 0743166X. DOI: 10.1109/INFCOM.2011.5934956.

[74] N. L. Van Adrichem, C. Doerr, and F. A. Kuipers, "Opennetmon: Network monitoring in openflow software-defined networks," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, IEEE, 2014, pp. 1–8.

[75] P. Sun, P. Ratul, M. Jennifer, and R. Princeton, "A Network-State Management Service," 2014.

[76] P. Megyesi, S. Molnár, A. Pescapè, A. Botta, and G. Aceto, "Available bandwidth measurement in software defined networks," *Proceedings of the 31st Annual ACM Symposium on Applied Computing - SAC '16*, no. i, pp. 651–657, 2016. DOI: 10.1145/2851613.2851727. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2851613.2851727.

[77] D. Comer and A. Rastegarnia, "Externalization of Packet Processing in Software Defined Networking," *IEEE Networking Letters*, vol. 1, no. 3, pp. 124–127, 2019. DOI: 10.1109/lnet.2019.2918155. arXiv: 1901.02585.

[78] *Apache Kafka*. [Online]. Available: https://kafka.apache.org.

[79] B. Chandrasekaran, B. Tschaen, and T. Benson, "Isolating and tolerating SDN application failures with LegoSDN," *Symposium on Software Defined Networking (SDN) Research, SOSR 2016*, 2016. DOI: 10.1145/2890955.2890965.

[80] R. Vilalta, J. L. de la Cruz, A. M. López-De-Lerma, V. López, R. Martínez, R. Casellas, and R. Muñoz, "UABNO: A cloud-native architecture for optical SDN controllers," *Optics InfoBase Conference Papers*, vol. Part F174-, pp. 4–6, 2020. DOI: 10.1364/OFC.2020.T3J.4.

[81] *kubernetes*. [Online]. Available: https://kubernetes.io.

[82] T. Kohler, F. Durr, and K. Rothermel, "ZeroSDN: A Highly Flexible and Modular Architecture for Full-Range Distribution of Event-Based Network Control," *IEEE Transactions on Network and Service Management*, vol. 15, no. 4, pp. 1207–1221, 2018, ISSN: 19324537. DOI: 10.1109/TNSM.2018.2873886.

[83] *Zeromq*. [Online]. Available: https://zeromq.org.

[84] S. T. Arzo, D. Scotece, R. Bassoli, D. Barattini, F. Granelli, L. Foschini, and F. H. P. Fitzek, *MSN: A Playground Framework for Design and Evaluation of MicroServices-Based SDN Controller*, 1. 2022, vol. 30, pp. 1–31, ISBN: 0123456789. DOI: 10.1007/s10922-021-09631-7.

[85] S. T. Arzo, R. Bassoli, F. Granelli, and F. H. Fitzek, "Multi-Agent Based Autonomic Network Management Architecture," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 3595–3618, 2021, ISSN: 19324537. DOI: 10.1109/TNSM.2021.3059752.

[86] M. van Steen and A. S. Tanenbaum, *Distributed systems*. 2020.

[87] I. Sommerville, *Software Engineering*, 9th ed. Harlow, England: Addison-Wesley, 2010, ISBN: 978-0-13-703515-1.

[88] *cobra*. [Online]. Available: https://www.omg.org/spec/CORBA/.

[89] G. Weiss, *Multiagent Systems*. The MIT Press, 2013, ISBN: 0262018896.

[90] A. H. Bond and L. Gasser, Eds., *Readings in Distributed Artificial Intelligence*, First. Morgan Kaufmann Publishers,Inc, ISBN: 093461363X.

[91] S. Russell and P. Norvig, *Artificial Intelligence A Modern Approach*, Third, c. Prentice Hall, 2010, pp. 1–4, ISBN: 9789004310087. DOI: 10.15713/ins.mmj.3.

[92] M. Wooldridge, *An Introduction to Multiagent Systems*, 2nd ed. Chichester, UK: Wiley, 2009, ISBN: 978-0-470-51946-2.

[93] H. S. Nwana, "Software agents: An overview," *The knowledge engineering review*, vol. 11, no. 3, pp. 205–244, 1996.

[94] "A Survey on Software Agent Architectures," *IEEE Intelligent InformaticsBulletin*, vol. 14, no. 1, pp. 8–20, 2013.

[95] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, *Swi-prolog*, 2010. arXiv: 1011.5332 [cs.PL].

[96] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the royal statistical society. series c (applied statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[97] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.

[98] *Knowledge Interchange Format*. [Online]. Available: http://www-ksl.stanford.edu/knowledge-sharing/kif/.

[99] *XML*. [Online]. Available: https://www.w3.org/TR/owl-xmlsyntax/.

[100] *OWL*. [Online]. Available: https://www.w3.org/OWL/.

[101] T. Finin, R. Fritzson, D. McKay, and R. McEntire, "Kqml as an agent communication language," in *Proceedings of the third international conference on Information and knowledge management*, 1994, pp. 456–463.

[102] "FIPA-ACL," [Online]. Available: http://www.fipa.org/repository/aclspecs.html.

[103] *REST*. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[104] *YANG*. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6020.

[105] V. Ribeiro, R. H. Riedi, R. G. Baraniuk, J. Navratil, and L. Cottrell, "pathChirp: Efficient Available Bandwidth Estimation for Network Paths," *Pam*, vol. 4, pp. 1–11, 2003. DOI: 10.2172/813038.

[106] "Schemes to measure available bandwidth and link capacity with ternary search and compound probe for packet networks," *LANMAN 2010 - 17th IEEE Int. Work. Local Metrop. Area Networks*, pp. 1–5, 2010. DOI: 10.1109/LANMAN.2010.5507150.

[107] G. Aceto, V. Persico, A. Pescapé, and G. Ventre, "SOMETIME: Software defined network-basec Available Bandwidth measurement in MONROE," *TMA 2017 - Proc. 1st Netw. Traffic Meas. Anal. Conf.*, 2017. DOI: 10.23919/TMA.2017.8002918.

[108] D. J. Hamad, K. G. Yalda, and I. T. Okumus, "Getting traffic statistics from network devices in an SDN environment using OpenFlow," 2016.

[109] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "PayLess: A low cost network monitoring framework for software defined networks," *IEEE/IFIP NOMS 2014 - IEEE/IFIP Netw. Oper. Manag. Symp. Manag. a Softw. Defin. World*, pp. 1–9, 2014, ISSN: 1542-1201. DOI: 10.1109/NOMS.2014.6838227.

[110] M. Singh, N. Varyani, J. Singh, and K. Haribabu, "Estimation of end-to-end available bandwidth and link capacity in sdn," in *Ubiquitous Communications and Network Computing*, N. Kumar and A. Thakre, Eds., Cham: Springer International Publishing, 2018, pp. 130–141.

[111] S. R. Talpur and T. Kechadi, "A Forecasting Model for Data Center Bandwidth Utilization," in *SAI Intell. Syst. Conf.*, 2017, pp. 315–330, ISBN: 9781509011216. DOI: 10.1007/978-3-319-56994-9_22.

[112] Jarschel, M. Zinner, T. Hohn, T. T. Gia, and Phuoc, "On the accuracy of leveraging SDN for passive network measurements," in *2013 Australasian Telecommunication Networks and Applications Conference, ATNAC 2013*, 2013.

[113] S. Jung, C. Kim, and Y. Chung, "A Prediction Method of Network Traffic Using Time Series Models," in *Int. Conf. Comput. Sci. Its Appl.*, 2006, pp. 234–243. DOI: 10.1007/11751595_26.

[114] W. Yoo and A. Sim, "Time-Series Forecast Modeling on High-Bandwidth Network Measurements," *J. Grid Comput.*, vol. 14, no. 3, pp. 463–476, 2016, ISSN: 15729184. DOI: 10.1007/s10723-016-9368-9.

[115] Z. Jin, Y. Shu, J. Liu, Yang, and O.W.W., "Prediction-based network bandwidth control," in *2000 Canadian Conference on Electrical and Computer Engineering. Conference Proceedings. Navigating to a New Era (Cat. No.00TH8492)*, vol. 2, 2000, 675–679 vol.2. DOI: 10.1109/CCECE.2000.849550.

[116] B. Zhou, D. He, Z. Sun, and W. Ng, "Network traffic modeling and prediction with ARIMA/GARCH," *HET-NETs' 06 Conf.*, no. September, pp. 1–10, 2006.

[117] T. W. Cenggoro and I. Siahaan, "Dynamic bandwidth management based on traffic prediction using Deep Long Short Term Memory," *Proceeding - 2016 2nd Int. Conf. Sci. Inf. Technol. ICSITech 2016 Inf. Sci. Green Soc. Environ.*, pp. 318–323, 2017. DOI: 10.1109/ICSITech.2016.7852655.

[118] W. Jiang, "Internet traffic prediction with deep neural networks," *Internet Technology Letters*, vol. 5, no. 2, e314, 2022.

[119] D. Aloraifan, I. Ahmad, and E. Alrashed, "Deep learning based network traffic matrix prediction," *International Journal of Intelligent Networks*, vol. 2, pp. 46–56, 2021.

[120] J. Read, A. Bifet, B. Pfahringer, and G. Holmes, "Batch-incremental versus instance-incremental learning in dynamic and evolving data," in *International symposium on intelligent data analysis*, Springer, 2012, pp. 313–323.

[121] *otext*. [Online]. Available: https://otexts.com/fpp3/.

[122] L. Bottou, "Stochastic gradient descent tricks," in *Neural networks: Tricks of the trade*, Springer, 2012, pp. 421–436.

[123] J. Zhou, M. Tewariy, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat, "WCMP: Weighted cost multipathing for improved fairness in data centers," *Proceedings of the 9th European Conference on Computer Systems, EuroSys 2014*, 2014. DOI: 10.1145/2592798.2592803.

[124] R. Trestian, K. Katrinis, and G. M. Muntean, "OFLoad: An OpenFlow-based dynamic load balancing strategy for datacenter networks," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 792–803, 2017, ISSN: 19324537. DOI: 10.1109/TNSM.2017.2758402.

[125] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford, "Efficient Traffic Splitting on SDN Switches," *CoNEXT '15*, 2015.

[126] G. N. Senthil and S. Ranjani, "Dynamic Load Balancing using Software Defined Networks," *International Journal of Computer Applications*, pp. 11–14, 2015.

[127] S. Attarha, K. Haji Hosseiny, G. Mirjalily, and K. Mizanian, "A load balanced congestion aware routing mechanism for Software Defined Networks," *2017 25th Iranian Conference on Electrical Engineering, ICEE 2017*, pp. 2206–2210, 2017. DOI: 10.1109/IranianCEE.2017.7985428.

[128] J. Kurose and K. Ross, *Computer networks: A top down approach featuring the internet*, 2010.

[129] R. K. Jain, D.-M. W. Chiu, W. R. Hawe, *et al.*, "A quantitative measure of fairness and discrimination," *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, vol. 21, 1984.

[130] J. Lee, S. Bohacek, J. P. Hespanha, and K. Obraczka, "A study of tcp fairness in high-speed networks," *Tech. Rep.*, 2005.

[131] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993. DOI: 10.1109/90.251892.

[132] B. Prabhakar and R. Pan, "CHOKe A stateless mechanism for providing Quality of Service in the Internet,"

[133] W. Li, F. Zhou, K. R. Chowdhury, and W. Meleis, "Qtcp: Adaptive congestion control with reinforcement learning," *IEEE Transactions on Network Science and Engineering*, vol. 6, no. 3, pp. 445–458, 2018.

[134] Y. Kong, H. Zang, and X. Ma, "Improving tcp congestion control with machine intelligence," in *Proceedings of the 2018 Workshop on Network Meets AI & ML*, 2018, pp. 60–66.

[135] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *International Conference on Machine Learning*, PMLR, 2019, pp. 3050–3059.

[136] X. Nie, Y. Zhao, Z. Li, G. Chen, K. Sui, J. Zhang, Z. Ye, and D. Pei, "Dynamic tcp initial windows and congestion control schemes through reinforcement learning," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1231–1247, 2019.

[137] M. Yamazaki and M. Yamamoto, "Fairness improvement of congestion control with reinforcement learning," *Journal of Information Processing*, vol. 29, pp. 592–595, 2021.

[138] S.-J. Seo and Y.-Z. Cho, "Fairness enhancement of tcp congestion control using reinforcement learning," in *2022 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, IEEE, 2022, pp. 288–291.

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

[140] C. J. C. H. Watkins, "Learning from delayed rewards," 1989.

[141] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015, ISSN: 14764687. DOI: 10.1038/nature14236.

[142] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," pp. 1–9, 2013. arXiv: 1312.5602. [Online]. Available: http://arxiv.org/abs/1312.5602.

[143] S. Shapiro and T. K. Kissel, *Classical First-Order Logic*, ser. Elements in Philosophy and Logic. Cambridge University Press, 2022. DOI: 10.1017/9781108982009.

[144] *Resource Description Framework*. [Online]. Available: Resource%20Description%20Framework.

[145] *OWL guide*. [Online]. Available: https://www.w3.org/TR/2004/REC-owl-guide-20040210/#Introduction.

[146] T. R. Gruber, "Toward principles for the design of ontologies used for knowledge sharing?" *International Journal of Human-Computer Studies*, vol. 43, no. 5, pp. 907–928, 1995, ISSN: 1071-5819. DOI: https://doi.org/10.1006/ijhc.1995.1081. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1071581985710816.

[147] *hermit*. [Online]. Available: http://www.hermit-reasoner.com/.

[148] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang, "Hermit: An owl 2 reasoner," *Journal of Automated Reasoning*, vol. 53, Oct. 2014. DOI: 10.1007/s10817-014-9305-1.

[149] *Protege*. [Online]. Available: https://protegewiki.stanford.edu/wiki/Main_Page.

[150] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical owl-dl reasoner," *Journal of Web Semantics*, vol. 5, no. 2, pp. 51–53, 2007, Software Engineering and the Semantic Web, ISSN: 1570-8268. DOI: https://doi.org/10.1016/j.websem.2007.03.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570826807000169.

[151] B. Motik, R. Shearer, and I. Horrocks, "Hypertableau reasoning for description logics," *J. Artif. Intell. Res. (JAIR)*, vol. 36, pp. 165–228, Oct. 2009. DOI: 10.1613/jair.2811.

[152] S. Rudolph, "Oundations of description logics," May 2011.

[153] I. Mistrık, R. Bahsoon, N. Ali, M. Heisel, and B. Maxim, *Software architecture for big data and the cloud*. Morgan Kaufmann, 2017.

[154] *pade*. [Online]. Available: https://pade.readthedocs.io/pt_BR/latest/.

[155] *jade*. [Online]. Available: https://jade.tilab.com.

[156] *mango*. [Online]. Available: https://mango-agents.readthedocs.io/en/latest/.

[157] *fipaca*. [Online]. Available: https://www.dca.fee.unicamp.br/projects/sapiens/Resources/Agents/Platforms/FIPA/specs/fipa00037/XC00037H.pdf.

[158] *fipaip*. [Online]. Available: https://www.dca.fee.unicamp.br/projects/sapiens/Resources/Agents/Platforms/FIPA/repository/ips.html.

[159] *fipareq*. [Online]. Available: https://www.dca.fee.unicamp.br/projects/sapiens/Resources/Agents/Platforms/FIPA/specs/fipa00026/XC00026F.pdf.

[160] *fipaqu*. [Online]. Available: https://www.dca.fee.unicamp.br/projects/sapiens/Resources/Agents/Platforms/FIPA/specs/fipa00027/XC00027F.pdf.

[161] *fipamsg*. [Online]. Available: http://www.fipa.org/specs/fipa00061/XC00061E.html.

[162] *Osbrain*. [Online]. Available: https://osbrain.readthedocs.io/en/stable/.

[163] *Asyncio*. [Online]. Available: https://docs.python.org/3/library/asyncio.html.

[164] *Kytos*. [Online]. Available: https://github.com/kytos/.

[165] *mininet*. [Online]. Available: http://mininet.org/.

[166] *OpenAI Gym*. [Online]. Available: https://gym.openai.com.

[167] F. Ruffy, M. Przystupa, and I. Beschastnikh, "Iroko: A Framework to Prototype Reinforcement Learning for Data Center Traffic Control," *arXiv*, no. Nips, 2018, ISSN: 23318422. arXiv: 1812.09975.

[168] C. Amitabha, *sdwan-gym*. [Online]. Available: https://github.com/amitnilams/sdwan-rl.

[169] *riverml*. [Online]. Available: https://riverml.xyz/0.14.0/.

[170] *wireshark*. [Online]. Available: https://www.wireshark.org/.

[171] "On the self-similar nature of Ethernet traffic," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 25, no. 1, pp. 202–213, 2004, ISSN: 01464833.

[172] M. E. Crovella and A. Bestavros, "Self-similarity in World Wide Web traffic," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 24, no. 1, pp. 160–169, 2007, ISSN: 01635999. DOI: 10.1145/233008.233038.