An agent-based approach to disintegrate and modularise Software Defined Networks controller

Vijaya Durga Chemalamarri, Mehran Abolhasan and Robin Braun
School of Electrical and Data Engineering
University of Technology Sydney, Sydney, Australia
vijaya.d.chemalamarri@student.uts.edu.au

***Abstract*—The Software Defined Network paradigm deviates from traditional networks by logically centralising and physically separating the control plane from the data plane. In this work, we present the idea of a modular, agent-based SDN controller. We first highlight issues with current SDN controller designs, followed by a description of the proposed framework. We present a prototype for our design to demonstrate the controller in action using a few common use-cases. We continue the discussion by highlighting areas that require further research.**

***Index Terms*—Multi-agent systems, modular SDN controller, SDN, OpenFlow**

I. Introduction

Software Defined Networks separate and consolidate the data plane's control functions into a control plane. A data plane is a collection of interconnected forwarding devices, mostly switches. The control plane primarily consists of a core controller module to communicate with the data plane, Southbound interface(SBI), Northbound interface(NBI) and Northbound Applications(NBAs). While NBIs and NBAs are diverse, defacto SBI is the OpenFlow [1] protocol. Controllers use the OpenFlow(OF) protocol to communicate with forwarding devices.

A controller core handles connections with switches, receives messages from switches and registers them as events for further handling. The core dispatches messages to various applications. Network functions such as layer2 switching, layer3 routing, and load balancing are some examples of NBAs. NBI act as a communication channel between the controller core module and applications. Pyretic [2],Frenetic [3] and Procera [4] are some examples of NBIs.

The control plane instructs the OpenFlow switch on packet handling by exchanging messages. The switches store received instructions in local flow tables as flow rules, match all incoming packets against these rules, and execute corresponding actions. Without a matching flow rule, the switch forwards the packet to the controller for processing. NBAs running on the controller process the incoming packet, create a flow rule and configure this rule in the OpenFlow switch.

In *Physically centralised* control plane, controller and applications run on a physical node while managing switches. This setup quickly leads to scalability and security issues [5]. Ryu [6], NOX [7], NOX-MT [8], Floodlight [9] and Beacon [10] are examples of physically centralised controllers. In *Physically distributed* control plane, multiple control entities handle the data plane. These entities are either clustered or interconnected to share global information. Some of the examples of such controllers are ONOS [11], ONIX [12], HyperFlow [13] and OpenDayLight [14]. *Logically centralised* controllers such as HyperFlow, ONIX, ONOS, and OpenDayLight synchronise network-wide view amongst all controller instances using a variety of state dissemination mechanisms. *Logically distributed* controller architectures such as DISCO [15] have each controller instance managing a subset of the data plane and communicating relevant information with other instances. Distributed architectures need to deal with issues around consistency [5].

In this work, we present a modular, agent-based SDN controller. We first describe the motivation behind our work, followed by a description of the proposed framework. We then discuss two use cases and present a prototype to demonstrate the controller in action. We continue the discussion by highlighting areas that require further research.

Some issues with existing SDN controller architectures are:

A. Issues

1) *SBI Dependent:* OpenFlow [1] has become synonymous with SDN. Such an association is beneficial for standardising SBI; nevertheless results in severe dependency on a single SBI and hinders innovation. SDN applications and OpenFlow are heavily intertwined. The absence of a singular SBI to manage hybrid networks consisting of various forwarding devices, both OpenFlow switches and traditional network devices(wired and wireless), leads to multiple controller planes. By decoupling applications from OpenFlow, alternative applications and SBIs may be explored independently for a logically centralised control plane for hybrid networks.
2) *Flexible resource allocation and application isolation:* Monolithic software is a self-contained software application with tightly interdependent and coupled components that work as a single entity [16]. There is no flexible isolated allocation of resources to components in such a system. Unfortunately, most of the SDN controllers are monolithic by design [17] [18] [19] with tight coupling between applications and controller core. An organic organisation of such a system with diverse roles and functions is to separate the functions into communicating modules with dedicated resources.

3) *Application portability:* Consider POX [20] and Ryu [6] 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. Applications should be independent of the underlying controller platform.
4) *Code Redundancy and Reuse:* Most SDN applications perform a standard set of actions such as exchanging packets with data plane, parsing and un-parsing OF packets and storing network information creating redundant code. Such essential functions should be consolidated into modules allowing code re-usability.

Finally, by separating data from NBAs, knowledge can be formalised and reasoned using knowledge representation languages such as OWL [21].

B. Contributions

To summarise, existing SDN control plane architectures are monolithic processes. [16] defines *monolithic application* as a single, self-contained unit where components are interconnected and interdependent, resulting in a tightly coupled code. Popular controllers such as ONOS and ODL are monolithic [17] [18] [19]. Due to such tight coupling, monolithic software is heavily technology stack dependent and does not isolate resources leading to inefficient resource allocation. Most current SDN controllers are heavily tied with OpenFlow as SBI hinders the development and adaptation of alternate SBIs. Common applications such as L2 forwarding are not portable across controller platforms, thus consuming human effort to rebuild the same applications across multiple platforms.

An alternate approach for monolithic architecture is a modular design where software is decomposed into micro-services or agents. Modular controller design allows flexible allocation of resources such as CPU and memory to applications and controller core. The modular design also makes distributed deployment on edge devices and the cloud possible. A single component performs SBI-specific functions, and applications executing generic network functions irrespective of SBI can be built(e.g. pathfinding). In this work, we recast the controller into modules by consolidating network activities into agents. These agents communicate with other agents to perform a network function such as forwarding packets, handling congestion or addressing link failures while monitoring the network in parallel.

II. Architecture

We now describe a modular, agent-based SDN controller architecture MASDN (MultiAgent SDN) based on our earlier work [22]. A Multi-agent system is a group of homogeneous or heterogeneous agents operating cooperatively with or competing against each other [23]. Agents are autonomous and capable of perception and action in the environment. MASDN controller departs from monolithic design and consists of multiple social and rational agents. Each agent is isolated and communicates with other agents using Inter-Process Communication(IPC) or TCP. The controller core is an agent in the multi-agent system communicating with agents that run NBAs. Thus, NBAs no longer are tied to a specific SBI. The agent system can be deployed centrally on a single physical machine or distributed across multiple devices (physical, virtual (e.g. containers)) while using a shared knowledge base to maintain consistent information. Finally, multi-agent systems are a classic case of distributed artificial intelligence. We pave the way for a cognitive SDN control plane by enabling learning and reasoning capabilities in some agents. The proposed modular architecture is presented in Fig.1b.

By unwrapping the complexity of the SDN control plane, we categorise the controller's functionality (controller core, SBI, NBAs and NBI) into a set of essential functions. Broadly, the functions are a) connecting and maintaining connections with the data plane, b) computing flow rules and c) updating flow rules on the data plane. In SDN, the controller core maintains connectivity with the data plane, and NBAs compose flow rules based on their internal logic. NBAs themselves are tied up with OpenFlow protocol to receive and send packets. We thus start by disassociating applications from OpenFlow and consolidating application functions into agents. For instance, an SBI agent (in this case, OpenFlow agent) and data plane communicate using OpenFlow. Similarly, different agents implement different network functions for provisioning flows, building topology, maintaining infrastructure information, monitoring the network, maintaining a global network state, and managing the network to address network abnormalities such as a link down incidence and congestion, amongst others.

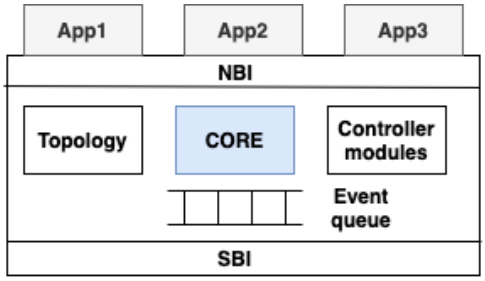
Thus, the agents in MAS SDN controller are categorised into

1) South Bound agents to interact with the data plane.
2) Provisioning agents to provision new flows, using L2 and L3 information.
3) Monitoring agents to monitor and maintain a network state in Knowledge Base.
4) Management agents to manage the network and handle network abnormalities by either rerouting flows to alleviate congestion and restore normality after a link failure or adjusting the TCP congestion window to maintain fairness amongst TCP flows and other management functions.
5) Knowledge layer stores a global network state in Knowledge Base.

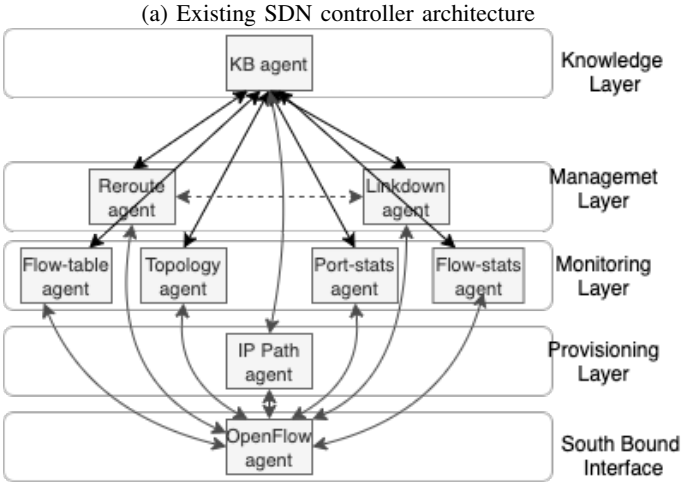
Fig.1b shows a high-level architecture of the agent system.

III. Agents and Interactions

We now present agents of the MASDN controller. The agents are broadly sub-grouped into layers based on their high-level functions. This agent system is expandable to include additional agents for network security and handling of non-OF devices.



(a) Existing SDN controller architecture



(b) Proposed agent-based modular architecture

Fig. 1: MASDN architecture

A. Southbound Interface Layer

This layer consists of agents that function as a southbound interface. Currently, this layer has an OpenFlow agent to handle OpenFlow packets.

1) OpenFlow agent

OpenFlow(OF) agent(OF_agent) is a TCP server establishing an OpenFlow communication session with the data plane. OF_agent maintains connections with the data plane, parses incoming OpenFlow packets and shares this information with other agents. OF_agent also receives messages from other agents, packs this information into OpenFlow packets and forward this information to the data plane. OF_agent runs two event loops catering to the data plane and the agent system for exchanging messages.

An OF_agent handles both OpenFlow and non-OpenFlow messages by identifying the recipient and forwarding the message to the intended receiver. The agent handles OpenFlow control messages such as hello, echo-request and feature-reply. In the case of data packet messages such as packet-in or port-status or messages to gather port and flow statistics, the agent relies on other agents for processing. Interacting with other/external agents to handle OpenFlow packets deviates from traditional SDN controller applications.

Traditional controller applications process and consumes OpenFlow packets. In contrast, the agent-based controller's OF_agent passes packet handling job to other agents in the agent system. While the reader might associate this behaviour with simply calling object methods, agents, unlike class objects, are autonomous - that is, their methods are not accessible to external entities and hence can reject or process a request. OF_agent's decision on where to forward the packet payload is based on the incoming packet's payload type. For instance, if packet-in message payload is a Link-Local Discovery packet(LLDP), the agent forwards this information to the Topology agent(Topo_agent).

Similarly, the agent forwards the non-QoS IP packet received due to the absence of flow rules (and the presence of a table-miss entry on the data path) to IP Path_agent for provisioning. Port-stats messages are forwarded to the Port-stats agent, and flow-stats messages are sent to the Flow-stats agent. Port-status message is sent to the topology agent.

In a few instances, the agent receives an unsolicited message from the data path, such as a change in a port's status or the removal of flows from the data path. A flow removed due to timeout does not necessarily need action or reconfiguration on the switch. Instead, the agent updates the knowledge base with this information. Hence, OF_agent forwards this packet to the flow table agent, which updates the knowledge base. If a port's status changes from 'UP' to 'DOWN', the controller needs to reroute all the affected flows without waiting for the flow rules to timeout, requiring the OF_agent to forward the information to link down the agent.

Separating OpenFlow protocol from applications into OF_agent is an effective way of eliminating the dependency of the SDN control plane on a singular SBI.

B. Provisioning layer

The primary function of agents in this layer is to proactively find paths to configure flow rules for a new flow. Agents performing switching, inter-VLAN routing, and load balancing constitute this layer.

IP Path_agent find paths to provision flows within a VLAN as in a traditional network. The agent computes *complete* end-to-end path and provides this information to OF_agent. *Complete* end-to-end path here refers to a set of all data paths and corresponding out-ports to configure flow rules. IP Path_agent employs Dijkstra's pathfinding algorithm to find paths between source and destination IP addresses. The agent periodically obtains a copy of topology from the KB_agent. KB_agent also pushes a new copy of topology to IP Path_agent upon any change in topology.

C. Monitoring layer

Monitoring agents are a group of agents responsible for monitoring the network and maintaining an updated network state. This layer can further extend to include other monitoring agents to monitor the health of OpenFlow data paths and other non-OpenFlow devices.

The Topology agent(Topo_agent) periodically(every fifteen minutes) requests OF_agent for updated topology. As mentioned earlier, OF_agent uses LLDP to discover topology and

relays back to Topo_agent. While a 'UP' to 'DOWN' port status change does not trigger a complete discovery, a status change from 'DOWN' to 'UP' triggers Topo_agent to initialise a complete link discovery.

Other agents in the monitoring layer are (a) the Port-stat agent(Ps_agent) for monitoring and collecting port counter information of all ports every fifteen secs,(b) the Flow-stat agent(Fsa_agent) for querying flow statistics of flows provisioned on a data paths' ports and (c) Flow-table agent(Fta_agent) to receive flow-removed message and update flow's information in the knowledge base. While port statistics are stored in the knowledge base, flow statistics need not be stored in the knowledge base, instead are only used by the rerouting agent to estimate the flow demand.

D. Management layer

While the agents in the monitoring layer collect, process and store network information, agents in the management layer consume this information to manage the network during a network abnormality. Currently, we have implemented agents to handle congestion and link failure.

Link-Down agent(LD_agent) addresses link failures by updating port and neighbour information in the knowledge base.

Reroute agent(RR_agent) is notified by KB_agent(managing the knowledge base) of possible network congestion. RR_agent then identifies a subset of affected flows to reroute on alternate paths. The agent reasons the flow rerouting problem as a constraint satisfaction problem, thus eliminating the possibility of shifting network congestion to other network parts. We have discussed the agent's operation in our previous paper [24].

E. Knowledge

Agents in the agent system need a formal representation of language for communication. An ontology of a domain is a formal specification of concepts and relations in a domain while defining vocabulary for interaction and implementation [25].

Formal representation of knowledge allows automated reasoning, where reasoning is deducing new facts from existing facts while identifying inconsistencies. Object-oriented programming handle facts based on the domain-specific rules programmed as methods and are incapable of reasoning or identification of inconsistencies in data [26]. KB_agent captures information about switches, links, ports, flows, flow rules, flow tables and properties such as a link down and congestion and flow association in the form of an OWL [27] ontology. To visually demonstrate components of the knowledge base, we use a handy tool- Protégé-2000 [28] to present a high-level ontology representation in Fig.2.

F. Agent interactions

We now discuss interactions between agents and the information they exchange in the context of three fundamental network functions - provisioning, maintaining, and monitoring the network. Messages exchanged between agents are of format

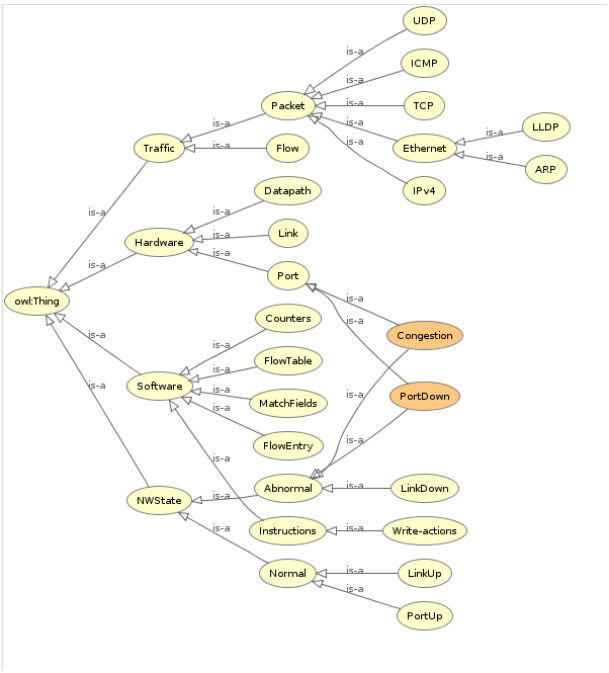


Fig. 2: Knowledge Base

$< messagetype >::< information >$. Please note that the agent system can include additional agents and is not limited to only these functions.

1) Provisioning flows

OF_agent receives an packet-in message from the data plane. Upon parsing the packet, relevant information(e.g., Source IP and Destination IP) is passed on to the IP Path_agent in format *path::(srcaddr,dstaddr)*. IP Path_agent uses a recent copy of topology to compute a path and responds to OF_agent in format $path::(srcaddr, dstaddr) :: [(dp_a, pr_a), (dp_b, pr_b), ..., (dp_n, pr_n)]$ where srcaddr is source address, dstaddr is destination address, (dp_i, pr_j) is (dpid,outport) tuple of intermediate data path.

Every new flow configured triggers IP Path_agent to push this information to the knowledge base. Agent interactions while provisioning a flow are shown in Fig.3.

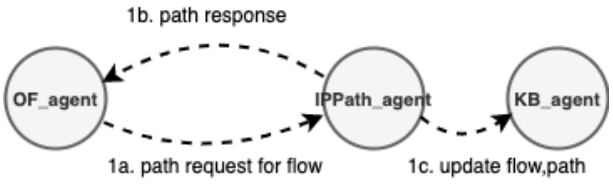


Fig. 3: Provisioning a new flow

2) Monitoring network

Agents in the monitoring layer periodically trigger OF_agent and gather network information. Network information includes various ports' operational status, port counters and topology information. Topo_agent requests OF_agent to

discover links; OF_agent, in turn, generates a Link Layer Discovery Protocol(LLDP) packets and relays this information back to Topo_agent. Similarly, OF_agent relays port_counters gathered back to Ps_agent. In Fig.4, we use 'mon_agent' as a generic name for all agents in the monitoring layer.

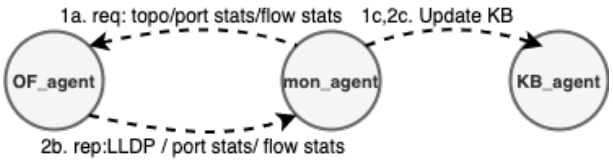


Fig. 4: Monitoring network

3) Manage network resources

Two common network abnormalities are a) link failure and b) congestion. In case of congestion, there is no explicit notification from the data plane; KB_agent identifies and flags congested links to RR_agent. In Fig. 5a, one can see KB_agent in action. Upon provided with a fact 'has TxBytes = 91234' , the KB_agent infers congestion(highlighted in yellow).

Upon receiving information from KB_agent, RR_agent fetches flow stats by querying OF_agent and attempts to reroute flows to an alternate path within the constraints of the available link bandwidth. Unlike network congestion, a link-down incident is registered when LD_agent receives a port-down message from OF_agent. LD_agent updates the internal network state stored in the knowledge base, as shown in Fig.6a. LD_agent fetches affected flows from the knowledge base and reroutes affected flows. It has to be noted that the system re-configures flows upon receiving a port-down message and not a packet-in message.

IV. Implementation and Testing

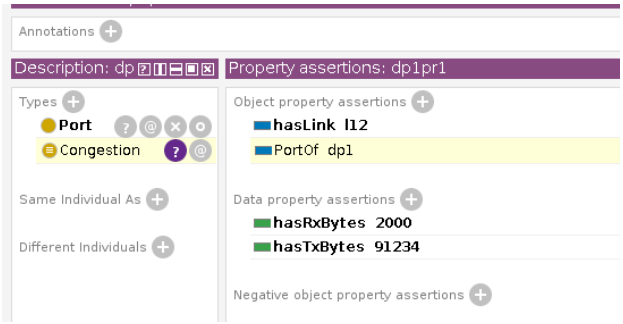
The use-cases aim to demonstrate a multi-agent controller in action as a centralised and distributed controller.

A. SetUp

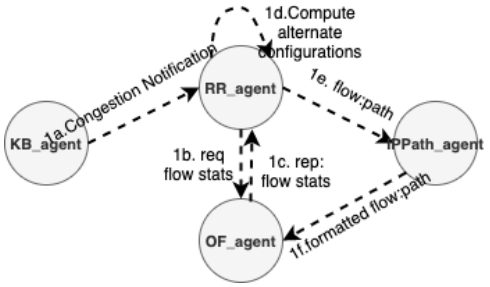
This section presents a prototype of the proposed multi-agent controller, MASDN. We chose Osbrain for ease of quick prototyping. Osbrain [29] is a python based multi-agent platform.

Agents use messaging queue- ZeroMQ [30] and Pyro4 [31] for communication. When deployed on a single physical node, the agents communicate via Inter-Process Communication(IPC), otherwise use TCP in multi-node environments. OF_agent is a multi-threaded agent, running a ZMQ communication loop and an asyncio [32] event loop on dedicated threads. While the ZMQ loop facilitates inter-agent messaging, the asyncio event loop connects and handles incoming OpenFlow messages from the data plane. OF_agent uses a lightweight python library PyOF [33] to parse OpenFlow packets. This prototype uses OpenFlow v.1.3.

Since the MASDN controller can be centralised or distributed, we tested both variations of deploying the controller. Fig.7 shows multi-node deployment where the agents are

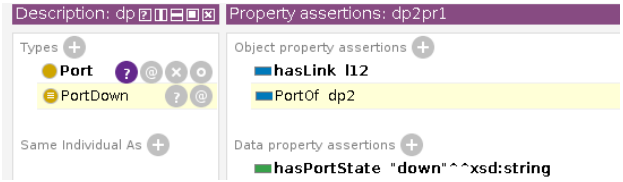


(a) KB_agent inferring congestion



(b) end-to-end communication

Fig. 5: Handling congestion



(a) KB_agent inferring a port is down

(b) end-to-end communication

Fig. 6: Handling Link down events

deployed on individual Raspberry Pi Model B [34] devices. We emulated the topology in mininet [35].

B. Tests

We measured flow-provisioning latency using packet capture timestamps in Wireshark [36]. To measure controller response time(in milliseconds), we triggered five TCP flows and measured the flow-setup duration as the time elapsed between each flow's incoming packet-in message and outgoing flow-mod message. The flow setup times for a set of twenty-five runs for the modular controller, when configured on a single machine(PS_m), on Raspberry Pis(PS_r), and Ryu(Ryu), are captured in Fig. 8a.

Ryu performs relatively better than a modular controller. Though inter-agent communication contributes to latency, we also believe that enhancing OF_agent's design to handle multiple simultaneous connections can reduce latency. Our

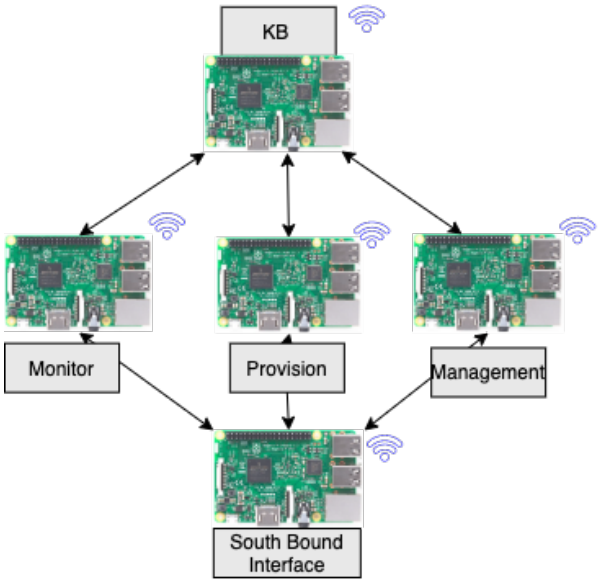


Fig. 7: Multi-node setup using Raspberry Pis

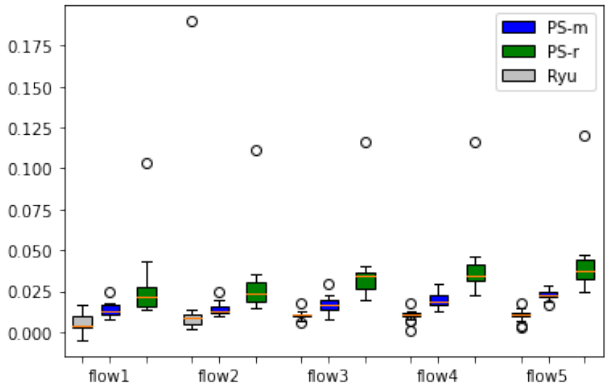
current scope of work does not presently address this. We will include it in our future work to improve the response time of the modular controller.

For the next test, Fig.8b, we calculated the time the controller took to re-provision flows in case of link failures. The latency is calculated based on the timestamp of the incoming port-down message and the corresponding flow-mod message sent by the controller.

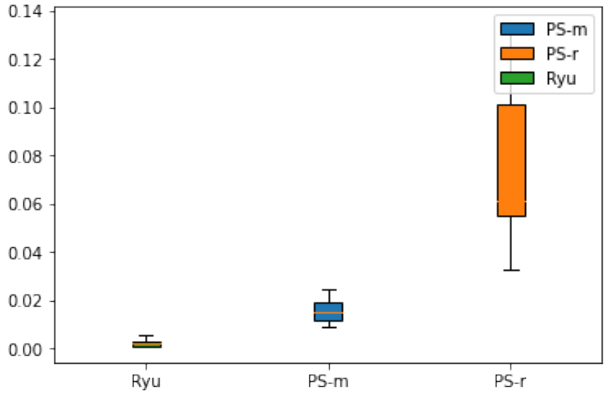
Though the MASDN prototype handles link failures proactively and steers traffic of affected flows at the prompt of the port-down message, latency is still an issue that must be addressed.

V. Related Work

We now discuss some existing work towards the control plane's disintegration. Most of the existing modular SDN control plane architecture designs adopt a micro-services approach, not an agent-based design like the proposed architecture. ZeroMQ [37] aims to distribute the control function by moving some of the control features back to the data plane. Network logic is split into lightweight control modules executed on individual processes. μONOS [38] proposes a new generation ONOS controller. The controller functions are decomposed as microservices and deployed on the cloud. μABNO [37] also proposes a microservices-based SDN controller for optical networks. Components synchronise using gRPC protocols. Comer [17] proposes micro-services-based controller architecture to outsource packet processing to an external entity using Apache Kafka [39]message distribution platform [40] propose multi-agent system-based autonomic network management is proposed with the actual implementation of agents left for future work. In our proposed work, we present a prototype and demonstrate a multi-agent-based SDN controller in action.



(a) flow setup time for five flows



(b) comparison for link-fail and recovery

Fig. 8: Preliminary test results

There has also been some recent work on the knowledge representation front for SDN. Authors of [41], [42] propose detailed ontology for SDN. Zhou [42] recursively fetches network information to build a knowledge graph and employs the SPARQL query engine to make inferences. [41] proposes services to update the knowledge graph and employs SPARQL to query knowledge. In the proposed work, we build a knowledge base to detect inconsistencies in ontology and network abnormalities such as link-down events and congestion events while leaving functions like pathfinding and resolving network abnormalities to agents.

VI. Discussion and Conclusion

This paper presented a novel way to build and deploy a modular SDN controller. We believe that an SDN controller redesigned as a multi-agent system provides the below advantages:

1) a step towards Cognitive SDN: By enhancing agents to learn, reason and communicate, we build intelligent agents and take a step towards building a Cognitive SDN control plane.
2) end-device and cloud-ready: A modular controller can be both physically centralised and distributed. In this paper, we have demonstrated both approaches. Furthermore,

we can containerise the agents as docker containers for Cloud deployment. Since the agents communicate using ZeroMQ, this is a potential deployment choice.

3) scalability and resources allocation: Adding new agents to this prototype is relatively straightforward. Allocation of dedicated resources to complex agents such as those employing machine learning techniques to classify traffic or predict loads is possible.
4) independent of OpenFlow: SBI layer can be extended to accommodate the NETCONF [43] agent to manage traditional networks in hybrid networks. All the agents in SBI layer can communicate with other agents in the agent system for decision-making.

A multi-Agent system does not come without challenges. Unlike monolithic systems, distributed systems inherently have higher latency based on the communication protocol. Also, agent synchronisation is essential to maintain a consistent network view. In our future work, we aim to address these issues.

REFERENCES

[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.

[2] Reich Jochua, "Pyretic: Modular SDN Programming," *Usenix*, vol. 38, no. 5, pp. 40–47, 2013. [Online]. Available: www.usenix.org

[3] 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.

[4] 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.

[5] 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.

[6] "Ryu SDN Controller." [Online]. Available: https://osrg.github.io/ryu/

[7] 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.

[8] 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

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

[10] D. Erickson, "The beacon openflow controller," ser. HotSDN '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 13–18. [Online]. Available: https://doi.org/10.1145/2491185.2491189

[11] 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, p. 1–6. [Online]. Available: https://doi.org/10.1145/2620728.2620744

[12] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, Others, and S. Shenker, "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

[13] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," ser. INM/WREN'10. USA: USENIX Association, 2010, p. 3.

[14] 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.

[15] K. Phemius, M. Bouet, and J. Leguay, "Disco: Distributed multi-domain sdn controllers," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, 2014, pp. 1–4.

[16] J. Ingeno, *Architecting Modern Applications*. Packt, 2018.

[17] D. Comer and A. Rastegarnia, "Toward Disaggregating the SDN Control Plane," *IEEE Communications Magazine*, vol. 57, no. 10, pp. 70–75, 2019.

[18] 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.

[19] 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.

[20] "POX." [Online]. Available: https://noxrepo.github.io/pox-doc/html/

[21] "OWL." [Online]. Available: https://www.w3.org/OWL/

[22] V. D. Chemalamarri, R. Braun, J. Lipman, and M. Abolhasan, "A multi-agent controller to enable cognition in software defined networks," in *2018 28th International Telecommunication Networks and Applications Conference (ITNAC)*, 2018, pp. 1–5.

[23] M. Wooldridge, C. Street, M. Manchester, and N. R. Jennings, "Intelligent Agents : Theory and Practice," *Knowledge Engineering Review*, no. October 1994, pp. 1–62, 1995.

[24] V. Chemalamarri, R. Braun, and M. Abolhasan, "Constraint-based rerouting mechanism to address congestion in software defined networks," in *2020 30th International Telecommunication Networks and Applications Conference (ITNAC)*. Los Alamitos, CA, USA: IEEE Computer Society, 2020, pp. 1–6. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ITNAC50341.2020.9315183

[25] 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. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1071581985710816

[26] S. Russell and P. Norvig, *Artificial Intelligence A Modern Approach*, 3rd ed. Prentice Hall, 2010, no. c.

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

[28] "protege." [Online]. Available: https://protege.stanford.edu/

[29] "osbrain." [Online]. Available: https://osbrain.readthedocs.io/en/stable/

[30] "zeromq." [Online]. Available: https://zeromq.org

[31] "pyro4." [Online]. Available: https://pyro4.readthedocs.io/en/stable/

[32] "asyncio." [Online]. Available: https://docs.python.org/3/library/asyncio.html

[33] "Kytos." [Online]. Available: https://github.com/kytos/

[34] "RPi." [Online]. Available: https://www.raspberrypi.org

[35] "Mininet." [Online]. Available: http://mininet.org

[36] "Wireshark." [Online]. Available: https://www.wireshark.org

[37] 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.

[38] "micro-ONOS." [Online]. Available: https://docs.google.com/document/d/1IZz_8EG1AII3JYmTYla585Gbpe9dfSwChO8lEkehp4A/mobilebasic

[39] "kafa." [Online]. Available: https://kafka.apache.org

[40] S. T. Arzo, R. Bassoli, F. Granelli, and F. H. P. Fitzek, "Multi-agent based autonomic network management architecture," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 3595–3618, 2021.

[41] C. Rotsos, A. Farshad, D. King, D. Hutchison, Q. Zhou, A. J. G. Gray, C.-X. Wang, and S. Mclaughlin, "Reasonet: Inferring network policies using ontologies." [Online]. Available: https://github.com/SemanticSDN

[42] Q. Zhou, A. J. G. Gray, and S. McLaughlin, "Seanet – towards a knowledge graph based autonomic management of software defined networks," 6 2021. [Online]. Available: http://arxiv.org/abs/2106.13367

[43] "netconf." [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6241