Towards In-Band Telemetry for Self Driving Wireless Networks

Prabhu Janakaraj*, Pinyarash Pinyoanuntapong*, Pu Wang, Minwoo Lee
Department of Computer Science
University of North Carolina Charlotte
Charlotte, USA
{pjanakar, ppinyoan, pu.wang, minwoo.lee}@uncc.edu

***Abstract*—Self-driving network is an emerging network automation design principle for building next generation autonomous networked systems based on machine learning algorithms trained on real-time experiences, i.e., network state measurements. However, existing network measurement techniques are designed on centralized architecture leading to considerable control overheads in wireless networks. In this work, we designed and implemented a distributed In-band network telemetry system (S-INT) and Wireless Network Operating System (WINOS) for self-driving wireless networks. On one hand, our proposed S-INT system significantly reduces network measurement overhead by embedding telemetry into flowing data traffic with a specialized packet header. WINOS system, on the other hand, seamlessly integrates programmable measurement, i.e., the proposed S-INT framework, with the programmable network control, while providing rich APIs to facilitate fast implementation of machine learning algorithms for intelligent and distributed network control. To show the effectiveness of our proposed system design, we implemented a multi-agent reinforcement routing as a traffic engineering application to optimize end-to-end delay performance. To the best of our knowledge, our implementation is the first one in the literature that enables multi-agent reinforcement learning algorithm to run on an actual physical wireless multi-hop network.**

I. Introduction

Why Self-driving Wireless Networks? Software Defined Networking (SDN) [1] brought more flexible network management functionality through a separation of network control and data planes. A global view of the network infrastructure can be obtained by the network control plane through a dedicated channel, hence enabling optimal resource allocation and traffic routing. SDN has been widely studied for wired networking environments like campus, data center, and wide area networks. Recent research works have explored SDN application in wireless networks such as campus WiFi, sensor networks, and cellular backbones [2], [3], [4]. However, only few works have considered wireless multi-hop networks [5], [6]. Wireless multi-hop networks, consisting of a mesh of interconnected wireless routers, have been widely exploited to build cost-efficient communication backbones, including wireless community mesh networks [7], [8], [9], high-speed urban networks (e.g., Facebook Terragraph network [10] and London small cell mesh network [11]), global wireless Internet infrastructures(e.g., Google Loon balloon network [12]), battlefield networks (e.g., rajant kinetic battlefield mesh networks [13]), and public safety/disaster rescue networks [14].

The coupling of programmable control using SDN with the inference capabilities of reinforcement learning promises unprecedented opportunities to realize next-generation self-driving wireless multi-hop networks, where network management and control decisions can be made in real time and in an automated fashion. Traditionally, network operators are driving the networked systems, who have to continuously develop and use scripts and tools to plan, troubleshoot, and optimize their networks. As user demands and network complexity grow dramatically, the traditional operator-driven networks are becoming inefficient. As an ultimate and ambitious goal for network management, self-driving networks, which draws an analogy to self-driving cars, automatically make network management and control decisions in real-time. The self-driving network can take as input a high-level goal related to performance or security (such as minimizing network congestion) and jointly determine (1) the measurements that the network should collect, (2) learning and inferences that the network should perform, and (3) the actions that the network should ultimately execute [15].

Challenges: As the enabling technology for self-driving networks, reinforcement learning (RL) algorithms are experience-driven optimization solutions that can generate optimal control decisions. In traditional wired SDN architecture, out-of-band centralized network telemetry approach is generally exploited to gather such experience because of the existence of a reliable dedicated control channel between the control plane and data plane. Network monitoring tools such as OpenFlow statistics, SNMP [16], sFlow [17], and NetFlow [18] are utilized to acquire key network telemetry data such as network topology, link delay, port status, queue delay, and link congestion by the control plane in network controller. By applying machine learning algorithms on the centrally available data it is possible to automate and optimize network.

However, due to the limited bandwidth and dynamic wireless conditions, wireless networks cannot be optimized in a centralized manner. Firstly, due to constrained network bandwidth and dynamic link conditions, employing a dedicated control channel is expensive. Secondly, existing centralized monitoring solutions are not capable of providing the real

This work is supported by NSF 1763182
* These authors contributed equally to this work

experience of the packet in wireless network. This demands the deployment of distributed reinforcement learning algorithms [19], [20], which in turn requires distributed In-band Network Telemetry (INT) systems. INT [21], originated from The P4 Language Consortium (P4.org), is a solution that enables collecting and reporting of network status, by the data channel (plane), without utilization and intervention of the control channel (plane). However, P4 INT requires additional hardware support.

The contributions of this paper are summarized as follows.

- We have designed and implemented S-INT, which is the first distributed in-band telemetry system proposed for wireless multihop networks, where each router runs its own telemetry module that is built on top of OpenFlow datapath/processing pipeline. Our preliminary experimental validation in wireless mesh testbed show that our proposed S-INT system is a cost-effective in-band network telemetry solution, which is very essential for enabling self-driving wireless networks
- We have developed a Wireless Network Operating System (WINOS), which is a distributed network controller running on each router. WINOS seamlessly integrates programmable measurement, i.e., the proposed S-INT In-band telemetry framework, with the programmable network control, while providing rich APIs to facilitate fast implementation of machine learning algorithms for intelligent and distributed network control.
- We have implemented a Multi-Agent Reinforcement Routing application for Wireless Mesh Networks using WINOS and S-INT. In particular, each router, acting as an agent, learns the optimal local traffic engineering (TE) policy in such a way that the collective TE policy of all routers can achieve the significantly improved end-to-end (E2E) TE performance in terms of delay, throughput, and packet loss. To the best of our knowledge, our implementation is the first one in the literature that enables multi-agent reinforcement learning algorithm to run on a physical wireless multi-hop network.

It is worth to note that this work aims to show that the proposed S-INT and WINOS can enable fast prototyping and evaluation of emerging RL algorithms for wireless multi-hop networks in the field. We are not attempting to show that our prototyped RL-based algorithms are the optimal solutions.

II. S-INT: Distributed In-band Wireless Network Telemetry

Self-driving networks are only feasible with accurate and timely feedback from the network elements. Existing network monitoring solutions increases network overhead dramatically and can only provide delayed experience of the packet, leading to deferred network control decisions. But, prompt response to network conditions are very critical for self-driving networks. In-band telemetry service can utilize data packets traversing the network ports for network metric measurement and transportation. However, existing in-band telemetry solutions are proposed for wired networks and centralized architectures [22], [23] [24], where in-band telemetry metadata increases the packet size by a significant order. Wired networks are capable of transporting fat packets of size 9000 bytes. Wireless networks are cannot handle fat packets. First, fat packets require additional wireless transmission time which will reduce the overall network utilization. Second, wireless networks can only have a maximum packet size of 2304 bytes. Thus, implementing the in-band telemetry system in wireless is challenging and every telemetry header can only have a specific metric.

Taking into account the practicability, we have designed and implemented S-INT, a distributed in-band telemetry system, where each router runs its own telemetry module, which is built on the top of OpenFlow datapath/processing pipeline. The proposed in-band telemetry system is enabled by three key components: new packet header called S-INT telemetry header, new packet matching actions: PUSH_INTL and POP_INTL, and the telemetry processor.

S-INT telemetry header: Network devices determines the encapsulation of the packet using a special field known as EtherType. In order to implement our S-INT telemetry system, first we defined the EtherType for the S-INT telemetry header and then implemented the encapsulation structure of the packet of size 16 bytes. Figure 1 shows the packet structure with our proposed header. Our proposed header structure is defined on the context of Ethernet frame, since OpenFlow bridge can only interpret Ethernet frames. Network application developers can utilize the fields within the header through our extended OpenFlow actions to gather the interested metrics. They can also specify sampling frequency, hop count, or even end-to-end datapath's as their constraints for measurement. In addition, we propose a template-based telemetry system where each telemetry template is unique and have their own measurement objective such as delay, bandwidth, and hop counts. Telemetry header consists of three fields for representing source datapath ID (the telemetry sender/source), destination datapath ID (the telemetry receiver/sink), and TLV field to specify which telemetry template is used. Each packet can only carry a single template due to the limitation of MTU size. However, in scenarios requiring to obtain two or more types of telemetry data we suggest to use alternate templates over a sequence of packets.

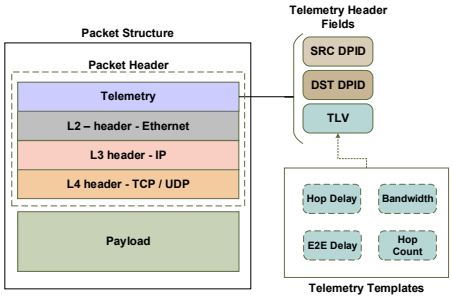


Fig. 1. Packet structure with S-INT Telemetry Header and Telemetry Template

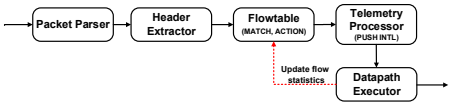


Fig. 2. Telemetry-enabled OpenFlow Datapath/Processing Pipeline: PUSH INTL performed by the telemetry sender

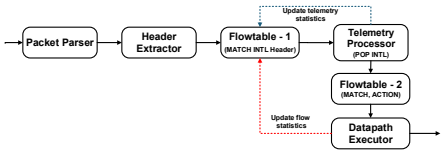


Fig. 3. Telemetry-enabled OpenFlow Datapath/Processing Pipeline: POP INTL performed by the telemetry receiver

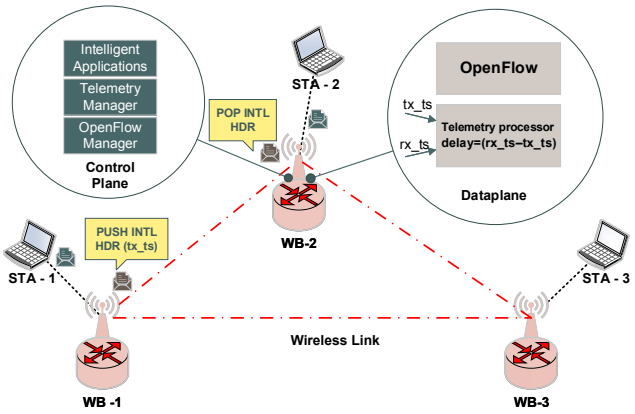


Fig. 4. S-INT Overall Architecture. Packet leaving WB1 contains the timestamped (tx_ts) and WB2 telemetry processor computes different between local timestamp (rx_ts) and packet timestamp (tx_ts) to get the delay

PUSH and POP Actions: OpenFlow protocol provides functions to encapsulate and decapsulate packets with headers such as MPLS, VLAN and so forth for data transport. We leverage the same functions to add and remove telemetry header to and from data packets. Figure 2 shows the data-path processing chain for PUSH telemetry header ACTION. At the sender's datapath, flowtable lookup is performed to identify the packet forwarding path based on MATCH and ACTION selection. If the path traversed by the packet is of interest for telemetry information, then the packet will have an additional action PUSH_INTL:template_type. This action appends the data packet with a new header and modifies the packet EtherType to local experimental EtherType. The above mentioned action is the last executed action by the datapath executor, before sending out the packet. Flow statistics of the respective flow is then updated by the datapath executor in terms of cumulative packet count and volume in bytes.

Datapath processing pipeline for POP action follows the sequence as in figure 3. Since local experimental EtherType is used as the identifier to know if the received packet contains telemetry header, at the receiver's datapath, flowtable MATCH is first performed to identify the EtherType. If it contains S-INT header, then the next ACTION to be performed on the packet is POP_INTL and then the EtherType is changed to IPv4 Packet. Packet is then handled in a normal dataplane processing pipeline, where a second flowtable lookup is performed for MATCH and ACTION. Datapath executor then forwards the packet following the ACTION and updates the flow statistics. Depending on the hop in which we implement the POP_INTL action, we can obtain 1-hop or End-to-End telemetry data. If we are only looking for 1-hop telemetry data, then we can even send the packets without any further encapsulation. However, if we are looking for 1-hop away or end-to-end telemetry data, then we need to encapsulate the packet further with transport headers such as MPLS.

Telemetry processor: Telemetry processor is the core of our S-INT framework. The PUSH_INTL and POP_INTL actions determine the operations performed by telemetry processor. If the action is PUSH_INTL:template_type, then the telemetry processor appends the header with the fields for the respective template type. If the received packet contains the telemetry header, upon identifying the telemetry template from the header simple arithmetic operations are performed with the telemetry data to retrieve link delay, bandwidth, and other network state information. After retrieving data, telemetry processor updates the telemetry statistics with new data.

In Figure 4, we illustrate an example case for measuring the link delay between two telemetry-enabled OpenFlow datapath's using our S-INT system. Consider host STA1 sends a packet to the host STA2. With OpenFlow rule as show in figure 5, WB1 adds telemetry header to the packet with the template type as delay using the PUSH_INTL action. Delay template appends timestamp (tx_ts) to telemetry header before sending our the packet. Once the packet is received by WB2 and telemetry header is removed with the POP_INTL action with template_type as delay. After retrieving the timestamp from the removed header, difference between current timestamp (rx_ts) on WB2 and packet header timestamp (tx_ts) is computed to get the delay experienced by the packet.

In order to realize the above mentioned illustration we should also extend the below listed datapath modules:

Packet Parser: Packet parser typically consists of the structure of every packet that exits in today's network including IPv4, IPv6, TCP, UDP, and ICMP. Once a packet is received on the switch port, the conformity of the packet is verified by checking its data structure. Non-conformed packets are automatically dropped. We extended the data structure of the packet library to include our custom header structure as in figure 1 to pass the verification stage.

Header Extractor: The header extractor identifies the fields within the received packet based on the packet type determined by packet parser. Every packet type has strictly defined header attributes with own data type. The header extractor scrutinizes the received packet and identifies the values for the specific header type attributes. For instance, if it receives a Ethernet packet then header extractor will identify the source and

OpenFlow rule on	MATCH	ACTION	STATS
WB1	in_port = 1	apply: push_intl, delay, out_port = 2	tx_pkts, rx_pkts, intl_delay
WB2	in_port = 2	apply: pop_intl, delay, out_port = 1	tx_pkts, rx_pkts, intl_delay

Fig. 5. S-INT: OpenFlow rules in Flowtable for delay estimation between Telemetry enabled OpenFlow datapath WB1 and WB2

destination Ethernet addresses. We extended this module with our new header structure and data type as shown in figure. 1.

Flowtable: Packet forwarding in SDN is handled based on the MATCH and ACTION instructions. The flowtable is a multidimensional row and column with depth of upto 1024. Typical structure of the flowtable is grouped into MATCH, ACTION and STATISTICS columns. MATCH columns will generally comprise of fields in L2, L3 and L4 packet headers in addition to datapath port numbers. ACTION column will contain the instruction of how to handle the matched packet. Typically, it can have a simple instruction such as forward the packet to a specified port, modify a packet, encapsulate or decapsulate a packet. STATISTICS columns are built with a handful of counters that identify statistics such as count and size of packets processed by the specific rule. To implement our telemetry templates, we extended the STATISTICS counters with additional fields to include telemetry metrics such as delay, bandwidth, and hop count. Figure 5 shows an example of flowtable structure with telemetry flows. Based on the chosen telemetry template, the resulting statistics fields will be varying. As a result, our PUSH_INTL and POP_INTL actions should be independent from the normal datapath processing action.

III. Wireless Network Operating System (WiNOS) with In-band Telemetry

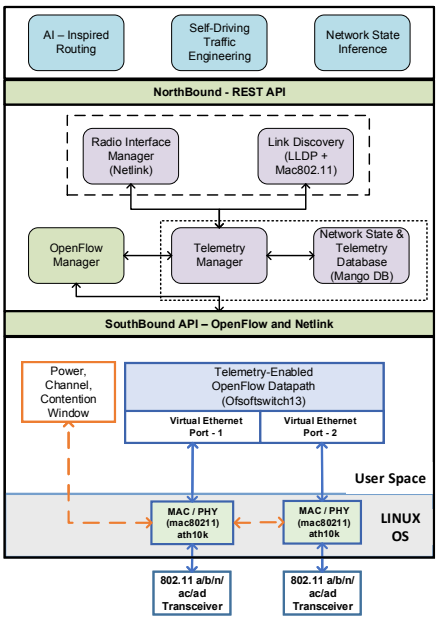


Fig. 6. WINOS with telemetry-enabled OpenFlow datapath

To implement S-INT telemetry system we need extensive modification on OpenFlow datapath and OpenFlow manager. In particular, OpenFlow datapath should be able to interpret telemetry data and OpenFlow manager should be able to collect and serve the data to other applications. As a result, we propose the distributed Wireless Network Operating System (WINOS), which runs on each wireless router and provides extended modules to realize the vision of self driving wireless networks. As shown in Figure 6, WINOS is designed to support the fast prototyping of AI algorithms for intelligent networking. It is built on the top of OpenFlow Manager based on RYU controller [25], telemetry-enabled datapath based on Ofsoftswitch13 [26] software switch, telemetry manager, network state and telemetry database based on MongoDB [27], and radio interface manager based on NetLink library.

OpenFlow Manager: SDN strongly relies on OpenFlow due to its simplicity of MATCH, ACTION and STATISTICS criterion. Although adopting the same principle for self-driving networks seems dauntingly useful, it should be capable to convey the experience of the network packets as results of followed ACTION. Hence, we propose to extend the OpenFlow STATISTICS module to incorporate telemetry data for conveying packet experience. OpenFlow1.3 protocol defines the structure of messages for representing the datapath elements such as ports, flowtable, packet, and so forth. OpenFlow enables switch interface to communicate with the controller using OpenFlow protocol. This imposes a requirement that any new functionality implemented on the datapath certainly requires modifying the OpenFlow protocol itself. In our case, to share the telemetry data from dataplane to the control plane, we have extended OpenFlow flow statistics message structure with our telemetry template fields such as SRC DPID, DST DPID and TLV fields.

Telemetry-enabled OpenFlow Datapath: To implement our S-INT telemetry framework, we modified the key modules of Ofsoftswitch13 according to the details in section II. Ofsoftswitch13 is a software switch that is designed based on the specifications of OpenFlow protocol version 1.3. This software switch runs entirely on userspace, thus making it the most suitable for prototyping new packet handling routines. The userspace switching leverages Linux TAP/TUN interface for integration with the operating system network stack. Ofsoftswitch13 supports attaching both virtual and physical ports to its datapath bridge. Packets received on these ports are processed through four key modules including packet parser, header extractor, flow table lookup, telemetry processor, and datapath executor as shown in Figure 2 and 3.

Telemetry Manager: Telemetry data access control and gathering process is handled by the telemetry manager. Any network application, which requires such telemetry data, sends the request to telemetry manager. After receiving the request, it checks Telemetry database and OpenFlow manager to identify if such data is already being gathered. If the application request type is new, then telemetry manager will identify the flows of interest and instructs the OpenFlow manager to disseminate OpenFlow rules to the corresponding datapath.

Network State and Telemetry database: Our network state database provides a RPC based interfaces for data access within kernel layer and also provides access interface via Northbound API's for network applications, such as reinforcement learning based routing algorithm. By having a database within the network controller, it is possible to enable stateful network applications such as firewall and also we can develop applications for data intensive traffic engineering applications. In our work, we extensively use mongoDB to store and serve the telemetry data. In addition, our state base also stores the network state such as link condition and topology formation in a time sequence manner.

Link Discovery Module and Radio Interface Manager: Network orchestration and control in SDN network highly rely on the gathered network topology information. In a fully closed/connected network such as wired networks, network topology is discovered using extended link layer discovery protocol (LLDP However, such network discovery mechanism fails in wireless networks. The primary reason for such failure is related to how the links and nodes are perceived in OpenFlow. In particular, direct adoption of the existing link discovery mechanism in wireless multi-hop networks will make all wireless nodes appear as if they are all 1-hop away from each other and they connect to a single port on the data plane. This complicates how we forward packets to the wireless nodes that are several hops away. Wireless channel is a broadcast medium and nodes accept packets that are transmitted with their wireless interface MAC address. Hence, we propose to extend the link discovery mechanism to be integrated with MAC80211 SoftMAC module[28]. MAC80211 itself contains discovery schemes that tells which nodes are connected to each other along with the link status. This combination enables network control and visibility in wireless multi-hop networks. In addition, with our integrated system we can control inherent wireless network properties such as link interference, transmit power, channel selection, topology formation and transmit contention window size.

IV. Multi-Agent Reinforcement Routing for Self-driving Wireless Mesh Networks

As one of the most important network management methods, traffic engineering (TE) aims to dynamically analyze real-time network traffic and perform optimal routing to meet the quality of service (QoS) requirements for the traffic flows. Optimizing the E2E TE metrics such as E2E delay and throughput is very challenging in wireless multi-hop networks due to the profound dynamics in traffic flow patterns, wireless link status, working conditions of wireless routers, and time-varying network topology. Recent advances in reinforcement learning (RL) have provided promising technologies for enabling experience-based model-free TE[19], [29], [20]. However, existing RL routing applications are based on non-realistic simulator or network emulator. In this section, we demonstrate a prototype of the self-driving wireless network by implementing a learning-based routing application on a WINOS-empowered wireless mesh network testbed. In particular, this routing application is based on our multi-agent reinforcement learning-based TE framework proposed in [20].

A. MDP for delay-optimal Traffic Engineering

The distributed traffic engineering (TE) can be formulated as multi-agent extension of Markov decision process (MA-MDP) for N routers [20], which is defined as a tuple of $\langle \mathcal{S}, \mathcal{O}_{1:N}, \mathcal{A}_{1:N}, \mathcal{P}, r_{1:N} \rangle$. In this MA-MDP formulation, the environmental states $\mathcal{S}$ consist of the network topology, the source and destination (i.e., source and destination IP addresses) of each packet in each router, the number of packets (queue size) of each router, and the status of links of each router. $\mathcal{O}_i$ defines the local observation of each router i. It contains the network state information, which is available at each router i. $\mathcal{A}_i$ is the set of actions that can be performed by router i. For our TE application, $\mathcal{A}_i$ contains the IDs of the next-hop neighbors of router i that router i can use as the next-hop forwarding node. P is the network state transition probabilities, which are generally unknown. r_i is the reward function of each router i, which is the (negative) 1-hop delay from router i to its neighbor. For each packet that enters the router i, the router needs to determine the forwarding action ($a \in A_i$) based on its local observation $o \in O_i$ of the network status. After the forwarding action is performed or the packet is sent out, the router will receive a reward r_i (i.e., the (negative) delay) when the packet arrives at its next-hop router $i+1$, which has its own local observation $o' \in O_{i+1}$. The return $G_i = \sum_{i=1}^{T} r_i$ is the accumulated reward (i.e., negative E2E delay) induced by forwarding a packet from its ingress router to its egress router. Each router selects forwarding actions based on a local policy π_i, which tells how the router chooses its action based on the observation. The policy can be stochastic by choosing an action according to certain probability or deterministic by choosing a fixed action. Our objective is to find the optimal policy π_i for each router so that the average E2E delay of all packets is minimized, i.e., the expected return $J(\boldsymbol{\pi}) = E[G_i|\boldsymbol{\pi}] = E[\sum_{i=1}^{T} r_i|\boldsymbol{\pi}]$ of the joint policy $\boldsymbol{\pi} = \pi_1, ..., \pi_N$ is maximized.

B. Multi-agent Off-policy Softmax RL Algorithm

To solve the above MA-MDP problem, we followed the multi-agent actor-critic (MA-AC) architecture [20]. In this case, each router has its own actor and critic running locally. The local critic uses exponential moving average to estimate the action-value functions $q_i^{\pi_i}(s,a)$, which criticize the action selections. Using critic's inputs, the actor improves the target policy towards the direction that can maximize the expected return,

Local Critic for Policy Evaluation: The performance of the policy π is measured by the action-value $q_i^{\pi}(s,a)$, which is a E2E TE metric. The action-value $q_i^{\pi}(s,a)$ of router i can be written as the sum of 1-hop reward of router i and the action-value of the next-hop router $i+1$, i.e.,

$$q_i^{\pi_i}(s,a) = E\left[r_i + q_{i+1}^{\pi_{i+1}}(s',a')\right]. \quad (1)$$

By applying exponential weighted average, the estimate of $q_i^{\pi_i}(s,a)$, denoted by $Q_i^{\pi_i}(s,a)$, can be updated based on

1-hop experience tuples (s, a, r_i, s', a') and the estimate of $q_{i+1}^{\pi_{i+1}}(s', a')$ of next-hop router, denoted by $Q_{i+1}^{\pi_{i+1}}(s', a')$, i.e.,

$$Q_i^{\pi_i}(s,a) \leftarrow Q_i^{\pi_i}(s,a) + \alpha[r_i + Q_{i+1}^{\pi_{i+1}}(s',a') - Q_i^{\pi_i}(s,a)] \quad (2)$$

where $\alpha \in (0, 1]$ is the learning rate.

Local Actor for Policy Improvement: Based on the estimated action-value, i.e., Q value, the local actor aims to improve the local policy towards the direction that can maximize the expected return $J(\boldsymbol{\pi})$. In this paper, we adopt the off-policy softmax algorithm as the local routing policy. In this case, the *target policy* the router aims to learn and improve is the greedy policy, i.e., selecting the action with the maximum estimated action-value. The *behavior policy*, which generates the actual actions for the learning agent, i.e., router, is softmax policy, where each action a is selected with a probability $P(a)$ based on the exponential Boltzmann distribution.

$$P(a) = \frac{\exp(Q_i^{\pi_i}(s,a)/\tau)}{\sum_{b \in \mathcal{A}_i} \exp(Q_i^{\pi_i}(s,b)/\tau)}$$

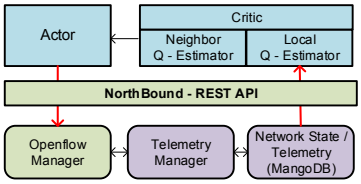


Fig. 7. multi-agent actor-critic reinforcement routing can be quickly prototyped as an application running on the WINOS of each router shown in Fig. 6

C. Learning Algorithm Implementation as an APP of WiNOS

The implementation of off-policy softmax learning algorithm is based on two northbound APIs provided by WINOS as shown in Fig. 7. The first API provided by network state database will provide the Q estimation for the local critic. Based on the Q value, the actor improves the target greedy policy, while generating the actual actions to be performed by router based on softmax policy. The actual action, i.e., next-hop router selection, is then forwarded to the OpenFlow manager based on another northbound API, Finally, the OpenFlow manager will translate the human-readable actual actions to the underlying OpenFlow instructions.

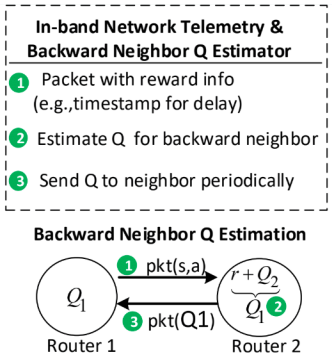


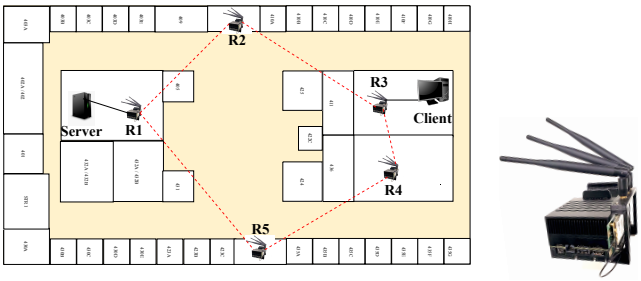
Fig. 8. Backward Neighbor Q estimation

Backward Neighbor Q Estimation: The key challenge to implement reinforcement routing algorithms is how to estimate the Q value without inducing so much control overhead. In particular, estimating Q value relies on the measurement of per-hop per-packet delay as shown in eq. 2. Directly requesting the per-packet delay information from the neighboring router could introduce significant overhead to the bandwidth-limited wireless channel. Therefore, it is necessary to redesign the way of exchanging information among neighbor. As illustrated in Fig 8, we propose the backward neighbor Q estimation for each router i, which aims to estimate the action-value of the (backward) neighbors whom the router i receives data from. The local Q estimation of router i is directly coming from its forward neighbors. The motivation of such design is based on the fact that the action-value Q_1 of predecessor router 1 is estimated based on the reward r_1 (i.e., per-packet delay) and the action-value Q_2 of current router 2. Both r_1 and Q_2 are immediately available at current router 2, instead of the predecessor router 1. Therefore, it is more cost-effective to let current router estimate the action-value of its backward router. Such scheme allows the action-value to be updated at the line speed, i.e., the speed at which packets come in the router and also reduce the overhead delay in control channel.

Implementation details: In our implementation, the local agent periodically observes the flow table from local controller REST API, which is provided by RYU rest_ofctl application. According to Fig. 8, when a packet from Router 1 is sent to Router 2, if there is a MATCH entry for the packet header fields, the forwarding rule at Router 2 is executed, which is defined by ACTION fields. In particular, MATCH fields contain [source destination mac, destination mac address, destination IP], where the destination IP of the packet is used as the local state/observation at each router. Based on such local state, the ACTION `set_field` (i.e., source destination mac, destination mac address and output port) is modified according to the softmax behavior policy, which needs the the local Q estimation that can be obtained from the forward neighbor router of the router 2. To stabilize the learning process, the ACTION fields are only updated for every N packets. At the same time, whenever router 2 receives a packet from the router 1, router 2 will keep updating the Q value of router 1 according to $Q_1^{\pi_1}(s,a) \leftarrow Q_1^{\pi_1}(s,a) + \alpha[r + Q_2^{\pi_2}(s',a') - Q_1^{\pi_1}(s,a)]$, where the reward r is the negative one-hop delay from router 1 to router 2. r is obtained by the proposed S-INT framework. Finally, router 2 will send the updated Q_1 back to Router 1 periodically via POST request. After Router 1 receives the Q_1 from router 2, it uses the new Q_1 as its local Q estimator.

V. Experimental Evaluations

Testbed Setup: Our experimental physical testbed consisted of 5 Nvidia Jetson Xavier nodes with Compex WLE900VX wireless interface card. We deployed our WINOS system on top of Ubuntu 18.04 Linux operating system running on each Nvidia Jetson node. Each wireless router was configured to operate on fixed 5Ghz Channel 36 and 40Mhz channel width in 802.11ac operating mode. We installed 5 wireless routers at



(a) (b)

Fig. 9. (a) Testbed deployed in a campus building (b) Wireless Router built on the top of Nvidia Jetson Xavier running S-INT and WINOS

various locations covering our lab floor area and 2 client hosts were connected at locations R1 and R3 as show in Figure 9. After the deployment, we manually inspected the topologies formed at every router and noticed that there were two possible paths from the client to the server. First, the upper bound path goes through R3 $\rightarrow$ R2 $\rightarrow$ R1 (2 hops) and lower bound path from R3 $\rightarrow$ R 4 $\rightarrow$ R5 $\rightarrow$ R1 (3 hops). A client sends a UDP traffic flow to the server with the varying traffic intensity of 1.0, 1.15 and 1.30 Mbps respectively following a Poisson distributed packet inter-departure time per second. The traffic flow lasts for 15 minutes. Each data traffic intensity keeps unchanged for 5 minutes and then the intensity is increased to next level. We use the average end-to-end throughput,the average end-to-end packet delay, and average end-to-end packet loss as the performance metrics. Since the wireless network conditions are dynamically changing overtime, we average the experiment results of 10 runs that are performed at different times (e.g., morning and night) of two consecutive days.

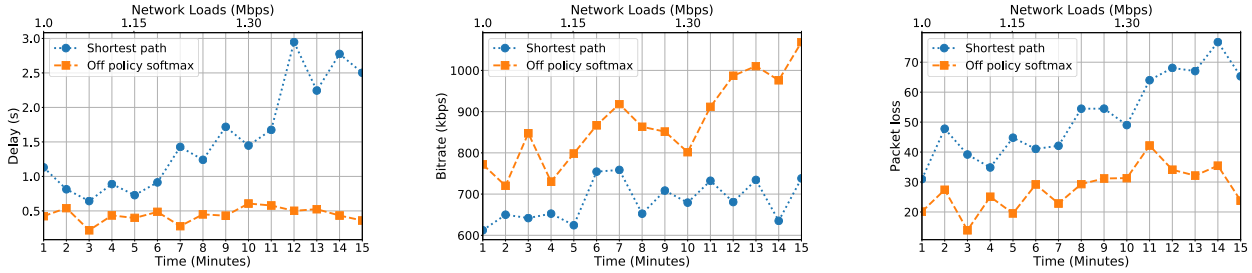
Learning-algorithms in the fields: The objective of this experiment is to show that the proposed S-INT and WINOS enable quick prototyping of the emerging reinforcement learning-based (RL) algorithms for wireless multihop networks in the field. We are not attempting to show that our implemented RL-based routing algorithms are the best solutions. Therefore, in this study, the naive shortest path (in terms of hop) was selected as a baseline to compare to off-policy RL algorithm with softmax action selection. As mentioned in [20], softmax action-selection, in general, helps the router to explore and exploit multiple routing paths to balance the traffic in high network loads and to reduce average packet delivery time. Thus, we selected softmax policy as a behavior policy for the agent and learning rate is set 0.1. Figure 10 shows the average delay, throughput, and packet loss rate for every 1 minute and the top x-axis shows the data traffic load increasing every 5 minutes. The dynamic network environment comes from the nature of the wireless medium (link delay) and increasing traffic load (queuing delay). The overall performances show the efficient routing policies of the learning-based TE since it is able to adapt to the non-stationary network environment and learn the optimal path dynamically. In term of the end-to-end delay, it can be seen that the off-policy softmax remains the same packet delivery delay as low as 0.5 second for the whole experiment. The shortest path routing performs poorly i.e., delay increases as traffic load increases. In Figure 10(b), the throughput of off-policy softmax surges up when higher data traffic (1.15 and 1.30 Mbps) is injected into the network. However, the throughput of shortest path increases only around 100 Kbps due to congestion, and the packet losses of shortest path routing continuously rise up to around 70 packets per second as shown in Figure 10(c).

Stability Analysis: The performance gap we observed in the previous section becomes even more evident when we evaluate the stability analysis over 10 runs. Figure 11 illustrates the mean and variance of the delay and throughput for each network load condition. Although the wireless network environment is heavily dynamic, the variances of delay and throughput of learning-based TE maintain low around 0.9 sec in term of delay and 100 Kbits in term of throughput. For example, even with highest network traffic (1.3 Mbits) injected into network, the average and variance of the end-to-end delay remains relatively small as 0.48 $\pm$0.81 for learning-based TE, while the shortest path routing experiences high average delay along with high delay variance (i.e., jitter). Similar phenomenon is also observed for the end-to-end throughput. In sum, as the traffic load grows, the learning-based TE algorithm leads to superior networking performance.

Overhead Analysis: The key feature of S-INT is to reduce the control overhead when the learning algorithms need to collect experiences for training. To observe the effectiveness of S-INT, we compare it to the probe-based measurement method, which sends extra probe packets to collect the network metrics such as link delay. The ICMP data packet is used to carry timestamp. Each router sends a timestamp probe packet after each data packet is sent out. In this way, the probe-based approach can achieve the same per-packet telemetry as S-INT. We evaluate the network delay and throughput performance under S-INT and probe-based telemetry respectively in Figure 12. X-axis represents the varying traffic loads, and y-axis shows the average of end-to-end delay and throughput for 5 minutes. The figure clearly shows that S-INT approach significantly reduces the control overhead and leads to much higher throughput and lower delay. This is because even with the small size (56 bytes) of probe packets, sending a probe packet out for every out-going data packet is very costly.

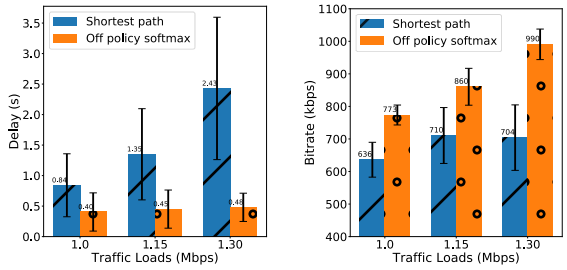
VI. Conclusion

In this paper, we proposed a distributed In-band telemetry system (S-INT) and a wireless network operating system (WINOS) for self-driving wireless networks. Our proposed system provides two key benefits (1) Programmable measurement using S-INT, resulting in low-overhead telemetry system and (2) Programmable wireless network control from WINOS for quick and easy implementation of AI-enabled distributed traffic engineering solutions such as Multi-Agent reinforcement routing. We implemented a traffic engineering application based on Multi-Agent Reinforcement routing on a physical wireless mesh testbed, using S-INT and WINOS systems. Our results show promising network performance



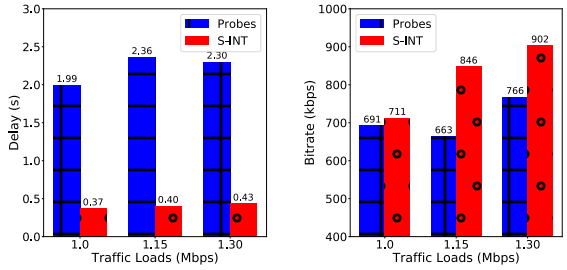
(a) E2E Delay (b) Throughput (c) Packet Loss

Fig. 10. Average of 10 runs under high network increasing load conditions, we measured the network metrics for every 1 minute



(a) E2E Delay (b) Throughputs

Fig. 11. Stability Analysis of off-policy softmax routing and shortest path routing



(a) E2E Delay (b) Throughput

Fig. 12. Performance comparison between probes and S-INT (single run)

in terms of delay, packet loss and throughput. We strongly believe, our proposed distributed WINOS and S-INT systems will open more research opportunities to realize Self-Driving wireless networks.

REFERENCES

[1] N. McKeown, "Software-defined networking," *INFOCOM keynote talk*, vol. 17, no. 2, pp. 30–32, 2009.

[2] I. F. Akyildiz, P. Wang, and S.-C. Lin, "Softair: A software defined networking architecture for 5g wireless systems," *Computer Networks*, vol. 85, pp. 1–18, 2015.

[3] L. E. Li, Z. M. Mao, and J. Rexford, "Toward software-defined cellular networks," in *2012 European Workshop on Software Defined Networking*. IEEE, 2012, pp. 7–12.

[4] K. Pentikousis, Y. Wang, and W. Hu, "Mobileflow: Toward software-defined mobile networks," *IEEE Communications magazine*, vol. 51, no. 7, pp. 44–53, 2013.

[5] A. Detti, C. Pisa, S. Salsano, and N. Blefari-Melazzi, "Wireless mesh software defined networks (wmsdn)," in *2013 IEEE 9th international conference on wireless and mobile computing, networking and communications (WiMob)*. IEEE, 2013, pp. 89–95.

[6] H. Huang, P. Li, S. Guo, and W. Zhuang, "Software-defined wireless mesh networks: architecture and traffic orchestration," *IEEE network*, vol. 29, no. 4, pp. 24–30, 2015.

[7] P. Frangoudis, G. Polyzos, and V. Kemerlis, "Wireless community networks: an alternative approach for nomadic broadband network access," *IEEE Communications Magazine*, vol. 49, no. 5, pp. 206 – 213, 2011.

[8] "New york city (nyc) mesh network," 2012, available: https://www.nycmesh.net/.

[9] "Germany freifunk wireless community network," 2003, available: https://en.wikipedia.org/wiki/Freifunk.

[10] "Facebook terragraph network," 2016, available: https://terragraph.com/.

[11] "London urban smallcell mesh network," 2017, available: https://www.thinksmallcell.com/Urban/city-of-london-deploy-urban-small-cell-mesh-network.html.

[12] "Google balloon powered global wireless internet," 2011, available: https://loon.com/technology/.

[13] "Rajant kinetic mesh networks for battlefield communication," 2018, available: https://rajant.com/markets/federal-military-civilian/.

[14] M. Portmann and A. Pirzada, "Wireless mesh networks for public safety and crisis management applications," *IEEE Internet Computing*, vol. 12, no. 1, pp. 18–25, 2008.

[15] N. Feamster and J. Rexford, "Why (and how) networks should run themselves," *arXiv preprint arXiv:1710.11583*, 2017.

[16] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "Simple network management protocol," STD 15, RFC 1157, SNMP Research, Performance Systems International, MIT ..., Tech. Rep., 1990.

[17] S. U. Rehman, W.-C. Song, and M. Kang, "Network-wide traffic visibility in of@ tein sdn testbed using sflow," in *The 16th Asia-Pacific Network Operations and Management Symposium*. IEEE, 2014, pp. 1–6.

[18] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better netflow," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 245–256, 2004.

[19] J. A. Boyan and M. L. Littman, "Packet routing in dynamically changing networks: A reinforcement learning approach," in *Advances in neural information processing systems*, 1994, pp. 671–678.

[20] P. Pinyoanuntapong, M. Lee, and P. Wang, "Delay-optimal traffic engineering through multi-agent reinforcement learning," in *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2019.

[21] J. Hyun and J. W.-K. Hong, "Knowledge-defined networking using in-band network telemetry," in *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2017, pp. 54–57.

[22] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, 2015.

[23] N. Van Tu, J. Hyun, and J. W.-K. Hong, "Towards onos-based sdn monitoring using in-band network telemetry," in *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2017, pp. 76–81.

[24] T. Mizrahi, G. Navon, G. Fioccola, M. Cociglio, M. Chen, and G. Mirsky, "Am-pm: Efficient network telemetry using alternate marking," *IEEE Network, accepted*, 2019.

[25] Ryu: Sdn controller. [Online]. Available: https://osrg.github.io/ryu/

[26] "Ofsoftswitch13," available: https://github.com/CPqD/ofsoftswitch13.

[27] "Mangodb," available: https://www.mongodb.com/.

[28] "Softmac," available: http://bit.ly/2Rq6vwG.

[29] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. Liu, and D. Yang, "Experience-driven networking: a deep reinforcement learning based approach," in *Proceedings of IEEE INFOCOM*, 2018.