INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2018, VOL. 64, NO. 2, PP. 209–216
Manuscript received March 31, 2017; revised April, 2018. DOI: 10.24425/119372

A Novel Software-Defined Networking Controller: the Distributed Active Information Model (DAIM)

Pakawat Pupatwibul, Ameen Banjar, Md. Imam Hossain, Robin Braun and Bruce Moulton

***Abstract*—This paper presents a new OpenFlow controller: the Distributed Active Information Model (DAIM). The DAIM controller was developed to explore the viability of a logically distributed control plane. It is implemented in a distributed way throughout a software-defined network, at the level of the switches. The method enables local process flows, by way of local packet switching, to be controlled by the distributed DAIM controller (as opposed to a centralised OpenFlow controller). The DAIM ecosystem is discussed with some sample code, together with flowcharts of the implemented algorithms. We present implementation details, a testing methodology, and an experimental evaluation. A performance analysis was conducted using the Cbench open benchmarking tool. Comparisons were drawn with respect to throughput and latency. It is concluded that the DAIM controller can handle a high throughput, while keeping the latency relatively low. We believe the results to date are potentially very interesting, especially in light of the fact that a key feature of the DAIM controller is that it is designed to enable the future development of autonomous local flow process and management strategies.**

***Keywords*—Distributed Networks, OpenFlow, DAIM Model, Software Defined Networking, Next Generation Networks**

I. Introduction

In the last few years, current needs and necessities of the modern world drive network technologies to be improved significantly in performance, complicatedness, functionality, and other aspects. Presently, networking has become very complicated to manage, control, activate, and monitor. This heterogeneity lead infrastructure to be increased in complicatedness which is hard to configure, maintain, re-engineer, recover and operate.

There is a need for an open and flexible architecture to implement the autonomic computing functionalities. Thus, Open Networking Foundation (ONF) has promoted a new network norm called Software-Defined Networking (SDN) architecture aiming to reduce complicatedness of management [1]. The main idea of SDN approaches is to separate functionality of data path from control path. The data path remains in a switch, whereas higher level routing decisions are separated in a commodity device called controller (a standard server) [2], [3]. The switch and controller can communicate using OpenFlow, which is considered as the first standard communications interface of SDN approach. OpenFlow defines messages such as received packets and sent packets [4], [5].

As advantages of SDN, companies get the programming ability to control the network with high scalability and flexibility, which can adapt quickly according to ever-changing circumstances. The layers of SDN structure include application, network operating system, and forwarding layers. Because network OS layer has APIs (Application Programming Interfaces) ability, it is possible to implement autonomic functionalities such as self-protection and self-optimisation.

SDN is very efficient at moving the computational load away from the forwarding plane and into a centralised controller. This centralisation brings optimality, but creates additional problems of its own including single-domain restriction, scalability, robustness, and the ability for switches to act autonomously.

Our new distributed active information model (DAIM) provides logical distribution of an SDN control plane, and should aid in the development of autonomous local processing within distributed networks. The DAIM model is a sustainable information model, which collects, maintains, updates and synchronises all the related information. DAIM offers adaptation algorithms embedded with intelligent agents and information objects to be applied to such complicated systems. Moreover, adopting DAIM model can manage complex systems in any distributed network, which is possible to be autonomous, adaptable, and scalable.

Section II reviews existing research on distributed OpenFlow controllers. In Section III, an overview of the DAIM model is described. Section IV addresses the main phases for developing DAIM. In Section V, the implementation of the DAIM model for SDN is discussed. Section VI illustrates the performance evaluation of the DAIM model. Section VII provides a summary and conclusion.

II. Related Works

This section reviews relevant prior research concerning SDN control planes. A distributed controller in this context is that SDN architecture uses more than one controller to control the data plane. The following approaches have employed different techniques to administrate data plane and manage the entire network state in distributed manner with scalability, simplicity, and global view. However, some of them have issues, and they may fail in arbitrary scenarios.

A. Kandoo

Scalability issues of the centralised model in SDN infrastructures motivate researchers to present Kandoo approach where it divided the control plane to tow levels [6]. One is a root controller with global applications, which are capable

Authors are with the Centre for Real-Time Information Networks (CRIN), University of Technology, Sydney, Australia (e-mail: {pakawat.pupatwibul, 11311103, 11469754, robin.braun, bruce.moulton}@uts.edu.au).

of accessing global network state. And the other level of controller remains close to the data plane called (local controller) which is capable of processing packets if the global view is not required. The idea was to bringing some of the control functionality to the data plane for better performance. Each controller is responsible for managing the network, for example, the root controller is responsible for the normal operation and the local controller can decide to redirect specific events to the root controller for processing or process events locally.

B. HyperFlow

HyperFlow is another approach to sorting the lack of scalability in a centralised domain of management. The authors enhanced SDN architecture by distributing the controllers into two components, one is controller's application to intercept events, and the other one is a middleware which is connected controllers together [7]. The motivation behind enhancement is to process event locally in the actual switch and configure commands for some other events that must be distributed. HyperFlow uses middleware for all communication between controllers which is able to Publish/Subscribe, publishers as senders of messages and subscribers as the receivers of messages. The controller's applications are useful in this approach where the idea here is not to sacrifice the simplicity of existed applications.

C. Onix

Onix enhanced the NOX controller with multiple contributions. It contains Network Information Base (NIB) which is an in-memory network graph for entities (network state), and there are two duplicated data stores, for exchanging information with NIB [8]. Onix provides a useful common programming API to build network applications. The controller is able to reflect the southbound API circumstances changes with northbound API, presented in NIB as management applications for network graph. Onix uses the OpenFlow protocol where the data plane indirectly modifies the NIB (through the controller). The controller has to guarantee that changes in the data plane are reflected in the NIB, and thus the NIB can change the data plane configurations.

All these approaches propose a different architecture and layer of SDN. Multiple controllers managing a network can bring many benefits such as enabling backup controllers to take over in the case of a failure, simplifying central view of the network, and reducing the look-up overhead by allowing communication with local controllers. However, distributing the control platform in Kandoo, HyperFlow, and Onix are not adapted to large data centres with several Autonomous Systems (AS), and they need extensive traffic among controllers to maintain a global network view. Moreover, there is also a potential downside related to trade-offs between staleness and consistency when distributing network state among the controllers. This may cause applications that believe they have an accurate view of the network to operate incorrectly.

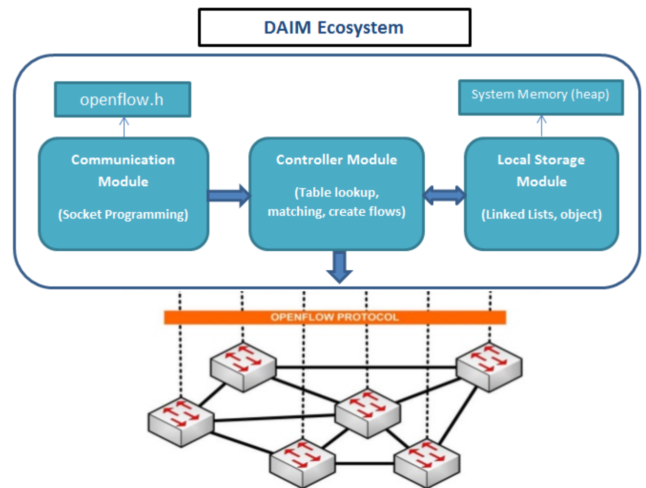

Fig. 1. DAIM Model Ecosystem

III. DAIM Model Overview

A new information model named: Distributed Active Information Model (DAIM) is presented to allow the local decision-making processes, which will essentially contribute to complex distributed network environments. An implementation of DAIM model is expected to introduce the requirements of the autonomic components of the distribution systems. The DAIM model can provide distributed systems with a sustainable information model, which collects, maintains updates and synchronises all the related information [9].

The DAIM model is implemented as an application on top of the OpenVswitch running in Mininet. DAIM uses the OpenFlow protocol to update forwarding tables in the local memory and switches. In the current design of DAIM, the control application executes as a thread on top of the Linux sockets. DAIM model uses a separate control channel to invoke commands between the controllers and switches. It is developed to support a cross-platform architecture, which has improved the variable types of running on both 32 and 64 bit CPU [10].

As can be seen in Figure 1, the DAIM ecosystem consists of three core modules namely communication module, controller module and local storage module working independently to achieve one single goal. The controller module is responsible for managing all the other modules so that the management and control can be traced to the controller module. Also, the controller module can actively support useful services for the other two modules. For example, the communication module provides routines for creating OpenFlow messages, and the local storage module provides storage information as well as the retrieval routines. Since all of the modules reside in a single process, the communication between each module is extremely fast.

The DAIM ecosystem uses OpenFlow protocol based on the OpenFlow switch specification version 1.0.0 (Wire Protocol 0x01). For the implementation of DAIM controller, the *openflow.h* is included in the header file to model the protocol and its defined messages as closely as possible. It is important

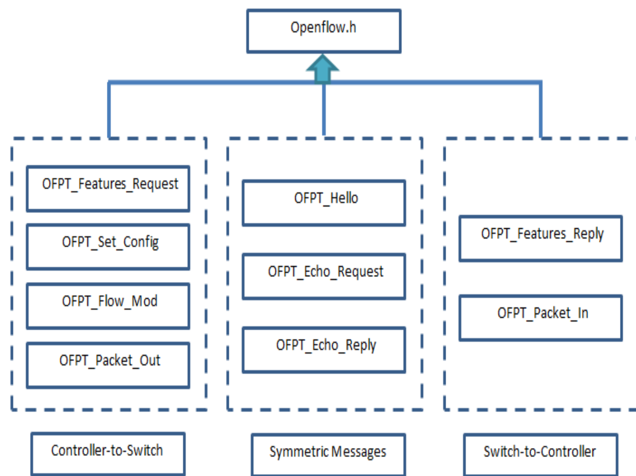

Fig. 2. Implemented OpenFlow Messages

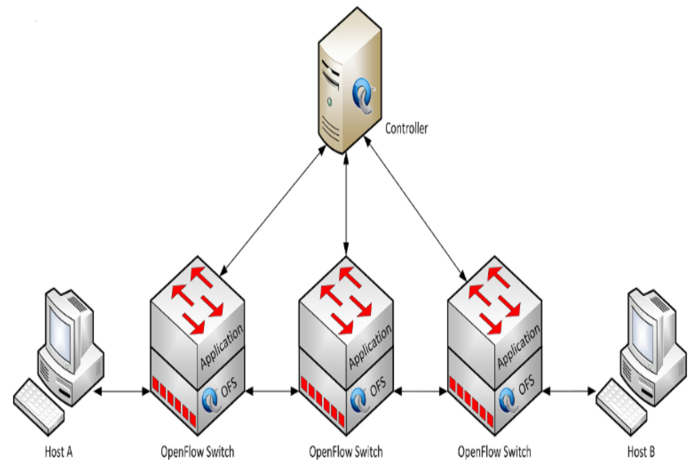

Fig. 3. DAIM Implementation Phase 1

to note that not all OpenFlow messages are implemented in the current state [11].

Inside the communication module, there are various routines for creating different types of OpenFlow messages. Figure 2 shows the implemented messages of the OpenFlow protocol in the DAIM model. The OFPT_Hello, OFPT_Features_Request, and OFPT_Features_Reply messages are implemented for the initialisation of the OpenFlow connection between DAIM controller and the switches. Echo request/reply messages are sent from either controller or switch and must return an echo reply. These messages are used to indicate the liveliness of a controller-switch connection and are repeated every 15 seconds. The controller uses an OFPT_Set_Config message to set the configuration parameters in the switch, whereas the OFPT_Packet_In message is used by an OpenFlow switch to notify DAIM of an unknown packet or to forward a packet to DAIM in the case of an associated action of a match. This message contains either the entire encapsulated packet or just the buffer ID of the buffered packet. The OFPT_Packet_Out message is a controller-to-switch message used by DAIM to forward a packet out of a specified port at the switch. Finally, DAIM can manage the flow table of a switch through the OFPT_Flow_Mod message type, which comprises the header match fields as well as the corresponding actions.

IV. DAIM Development Phases

This section describes the three phases of developing the DAIM model as follows [12], [13]:

A. Phase 1: Basic Carrier Functionality

In the first phase, DAIM model has been initially integrated to the SDN architecture by applying the implemented communication channel between the NOX controller and an OpenFlow switch. The DAIM model is developed in C++ using an open source NetBeans IDE and NetBeans Platform. In addition, DAIM model is created in different classes to represent each type of OpenFlow messages such as Packet-In, Packet-Out, Set-Config and Flow-Modification. The *openflow.h* is included in the header file to facilitate the implementation of such messages.

At this phase, DAIM is defined as a basic application channel used for message transmission in an OpenFlow network. No control functions have been implemented. Essentially, this phase describes a simple communication channel based on a client-server model (see Figure 3). DAIM is developed using UNIX BSD socket programming API where the server socket connects to OpenFlow switch and the client socket connects to the NOX controller. The DAIM application listens on a particular port (default 6633) for messages from the NOX controller. The network architecture of OpenFlow is still the same, which has all of the high-level routing decisions made by the NOX controller, but will have the DAIM application processing and forwarding all OpenFlow messages (without any modification) from controller to switch and vice versa instead of the original secure OpenFlow channel.

B. Phase 2: Semi-Distributed Functionality

The structure of phase 2 is similar to the previous phase but includes some level of distributed event-based control plane for OpenFlow by distributing the DAIM controller to each connected OpenFlow switch to perform its functions locally (see Figure 4). A major distinction from the first phase is that the NOX controller gets replaced by each distributed DAIM controller. Now the DAIM controller will not only be a basic carrier for OpenFlow messages between switches and controller but also can gather information from the network and propagate its local MAC address table to act as an intelligent Layer 2 learning Ethernet switch. For example, it can store network information when connected nodes perform an ARP or ICMP session, and hence it is possible to forward flows directly from the switch according to the flow entries that are pre-defined by the DAIM controller. The algorithm for building an intelligent Layer 2 learning switch functionality consists of the following steps:

Algorithm 1 Ethernet Learning Switch

For each packet from the switch,

```
(1) Use MAC source and incoming port
    number to update the data structure
(2) if Ethernet frame type is LLDP(0X88cc)
(3) Drop the packet // Do not forward the
    link-local traffic
(4) else if MAC destination is multicast
(5) Flood the packet
(6) else if the output port is same as the input
    port
(7) Drop the packets
(8) else if the data structure contains a port
    for the MAC destination
(9) Forward the packet to the destination address
(10) else if the data structure does not contain
     MAC destination port
(11) Flood the packet
(12) else install the flow entry in the switch
     flow table
```

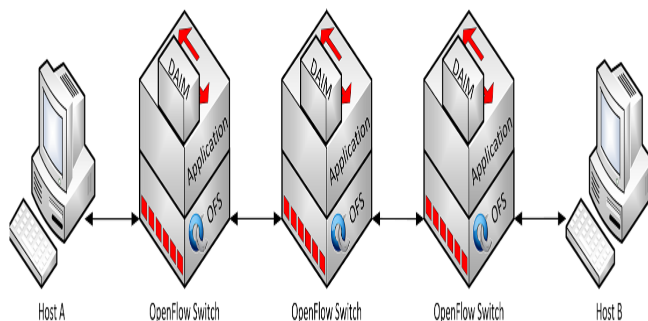

Fig. 4. DAIM Implementation Phase 2

At this stage, the DAIM controllers do not have a separate communication channel to exchange information between one another. The OpenFlow switches listen to its own connected DAIM controller on a specific port (e.g. 2000) for control messages. Another significant control function of DAIM in this phase is performing a certain process for querying network statistics (READ STATE protocol message). In addition, the DAIM controller is able to identify if a network problem happens, and also sends the corresponding message to the switch to update the ports and flow tables. In all cases, the connection is established per the TCP followed by, hello, and feature request/reply messages. This connection has to be developed in advance in order to process those functions.

C. Phase 3: Fully Distributed Functionality

Phase three aims to migrate all computational power to the DAIM model, which can manage each connected switch to produce some level of distributed computing network system. In addition, the implementation of this phase focuses on distributing the high-level decision making of traffic control to the DAIM controllers. Each distributed DAIM controller can actively share all information regarding its portion of the network to ensure fine-grained network wide consistency. For coordination purposes, DAIM controllers can also publish events as well as actively synchronise its local information with other associated controllers in order to construct the global network view. This distributed traffic management allows multiple levels of redundancy as each site has the ability to perform wide area system functions. Thus, each DAIM controller manages its affected switch and distributes useful information to other instances and if necessary communicates with the neighbouring domain.

V. DAIM Model Implementation

This section describes the software specification of the DAIM model, the implemented OpenFlow messages and modules. The implemented modules are comprised of the Communication module, the local Storage module, and the Controller module as well as the most significant messages that are needed for the communication between the switch and the DAIM controller. More details are presented in the following subsections.

A. Communication Module

The communication module is responsible for providing the communication mechanism and creating sockets of a two-way communication link between the controller and switch. In more details, it creates two processes for handling the communication between the switch to DAIM as well as DAIM to OpenFlow controller. The main purpose of these two processes is to forward messages both ways among the controller and switch and create signal handlers for notification of errors to the processes as well as the exiting of processes upon user's request. Sockets facilitate TCP/IP communication between two separate systems. Initially, sockets are created and then transferred over to the respective processes.

The client-server model is one of the most used communication paradigms in networking systems. Clients normally communicate with one server at a time. From a server's perspective and at any point in time, it is not unusual for a server to be communicating with multiple clients. A client needs to know the existence of the address of the server, but the server does not need to know the address of (or even the existence of) the client before the connection being established. Clients and servers communicate using multiple layers of network protocols in which this context will focus on the TCP/IP protocol suite.

The scenario of establishing connections between the server socket to receive the connection from the switch and the client socket to communicate with the controller is shown in Figure 5. Socket creation follows client and server model, where DAIM is the server socket for the switch and also is acting as the client socket to the OpenFlow controller.

The socket API and support for TCP and UDP communications between end hosts are described. Socket programming is the key API for programming distributed applications on the Internet.

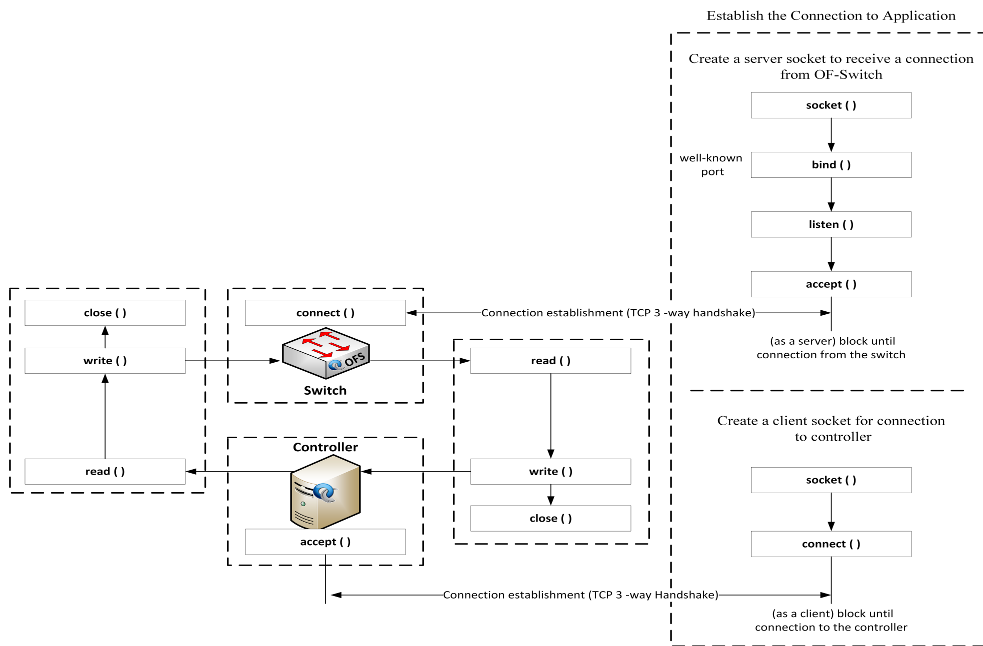

Fig. 5. Unix Socket Connection Setup

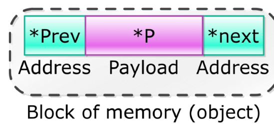

Fig. 6. DAIM Storage Block of Memory (Object)

B. Local Storage Module

The DAIM local storage module requires two main components including the Hosts and the Ports table. The information of network devices is stored in these tables, which are implemented using Linked list based storage. Linked list is a way to store data with structures so that the programmer can automatically create a new place to store data whenever necessary. Specifically, the programmer writes a struct or class definition that contains variables holding information about something, and then has a pointer to a struct of its type. Each of these individual structs or classes in the list is commonly known as a node.

To store network information into the storage table, we created a list of objects and cached information using blocks of memories (objects). Memory for the objects is then allocated by using C++ dynamic memory allocation methods. In linked list based storage, each object keeps the address of its preceding and subsequent objects. Therefore, each object can refer to its next and previous objects. The system's block of memory (object) is depicted in Figure 6.

The * prefix represents a pointer that holds the address of a memory block. Hence, the payload *p can point to any block of memory storing single or a combination of information such as host MAC address, switch data path ID, switch port, and IP address. The structure for the block of memory is defined by:

```
struct object
{
void *p;
struct object *next;
struct object *prev;
};
```

To store objects, we have created the object_lists (tables) structure in the storage module. The object_list is a representation of a table, and each of them includes the host entries object_list and the port entries object_list. Each object_list structure stores information such as the object size, address of the first object, address of the current object and address of the last object. Each object_list has a number of functions to manipulate the object_list itself. These functions can be used to add a new object, remove an object, free the memories used by the objects and retrieve an object from the list.

C. Controller Module

The controller module is responsible for maintaining the connectivity between the switches as well as allowing the switches to decide what actions to apply when a particular combination of network request is queued to the switch.

Firstly, the controller module establishes the connection to the switch using TCP/IP protocol stacks. To be able to receive the connection from an arbitrary OpenFlow switch, the controller module creates a server socket that listens for any incoming connections from the switch.

Upon successful connection to the switch, the controller module goes in a sequential mode of operations by using the "communicate_with_switch" method, which performs the two functions in order, read a message from the switch and send a respective reply message to the switch. The methods that reside within the "communicate_with_switch" are read_from_switch and send_information. Depending on the

nature of the messages received from the switch, the controller then decides and directs the switch to perform the follow-up actions.

When the switch receives a flow's first packet, it will be sent to DAIM application because there is no flow entry in the switch's flow table to match this flow. The DAIM controller module determines the action of switch packet forwarding upon receiving the OpenFlow OFPT_Packet_In message from the switch (see Figure 7). From the Packet_In OpenFlow message, the controller module first checks whether the packet is an ARP type by analysing the encapsulated Ethernet II frame type.

From the Ethernet II frame, the controller module then parses the MAC address of the source host as well as its IP address and adds this information into the respective tables if they are not already stored in the tables. After the MAC addresses are learned, if the destination MAC address in the Ethernet II frame is a broadcast type, then the controller module sends a packet_out message to the switch with the forwarding action of flooding the packet to all switch ports except the ingress port. If the destination MAC address is not broadcast, the controller module then looks up for the host MAC address in the table, and if found it will send an ARP flow modification message with an action to create a new flow entry in the switch flow table. This flow entry will regulate the forwarding of all future packets by matching the incoming packet's source and destination addresses.

Furthermore, if the packet_in message of Ethernet II frame is an IPv4 type, the controller module parses the protocol type from the IP header frame and then follows the same procedure as ARP Ethernet type. For TCP and UDP protocol types, the controller module creates flow modification messages incorporating the TCP and UDP source and destination ports.

VI. DAIM Model Performance Evaluation

A performance analysis of the DAIM controller was conducted using the Cbench open benchmarking tool. A built-in utility was used as a generic software framework to allow the development of tests for the DAIM controller.

A. Test Bed Description

The goal was to enable side by side comparisons of pre-existing OpenFlow controllers with the DAIM controller. The three pre-existing controllers used in the tests were NOX, POX and NOX-MT. For all experiments, each controller implements a normal L2 learning switch application provided by the controller. For every switch on the chosen path, the switch application performs MAC address learning. The packets get sent out of the last port where the traffic from the destination MAC address has arrived. Packets with an unspecific destination are flooded. All setups were run on an Intel® Core(TM)2 Duo CPU E8400 running at 3 GHz with 4 GB of RAM. Mininet was installed with Ubuntu 12.04.5 LTS x86_64 and a Linux 3.5.0-54-generic kernel.

B. DAIM Communication Channel Results

Throughput: This test shows the measurements of the throughput of each controller. The default Cbench configurations of test loops and duration are used under this test. Also, the mappings of destination MAC addresses are learned before the test.

Figure 8 shows the Cbench throughput results of the extended OpenFlow controllers with a single thread, in which an OpenFlow switch is contacting the controller in response to a new Packet_In (new packet arrival) event. The number of Flow_Mod responses per second for 16 tests are plotted.

Each test was run with a 1,000 ms duration and 100,000 unique-source MAC addresses. DAIM channel running with NOX is able to handle the highest throughput on average 23,196 Flow_Mod responses per second, followed by NOX with average 20,540 responses per second. In comparison, the Python-based controller ran slower. The throughput performance of a DAIM channel running with POX could serve on average 6,438 responses per second. In comparison, POX achieved 5,952 responses per second on average. These results suggest that connecting the DAIM channel to either NOX or the POX controller enabled higher throughput than the original controller.

Latency: This test looked at the average latency introduced by different controllers, and compared the latency when running with the DAIM channel. The results should be understood in light of the fact that there are differences in the implementation of the switch in the POX and NOX controllers.

The Cbench tool was used to perform the latency test, in which an OpenFlow switch forwards a packet to the controller and waits for a reply, then repeats this process as quickly as possible. The total number of responses received within a fixed period was used to calculate the average time the controller took to process each event. The results are shown in Figure 9. First, it can be seen that the (Python) POX controller runs slower than the (C++) NOX controller. It can also be seen that the DAIM channel running with NOX has the lowest average latency at 56.79 μs. The average response time for the DAIM channel running with POX is 149.98 μs. The POX controller when run alone had the greatest latency: 179.03 μs on average. These results suggested that adding the DAIM channel reduced the latency of both the NOX and the POX controllers.

C. Layer 2 Learning Switch Application Results

Throughput: This test looked at the average maximum throughput of three pre-existing controllers, and compared them with DAIM.

Figure 10 shows Cbench throughput results. In this test, DAIM, NOX and POX were used to process flows in a single threaded manner. These controllers and Cbench are each bound to a distinct physical core of a processor. NOX-MT, however, is a highly optimised multi-threaded implementation of NOX, and hence, not surprisingly, NOX-MT shows the best throughput (292,612 responses per second). Of the non-multithreaded controllers, DAIM produced the highest throughput, at 162,519 responses per second, followed by

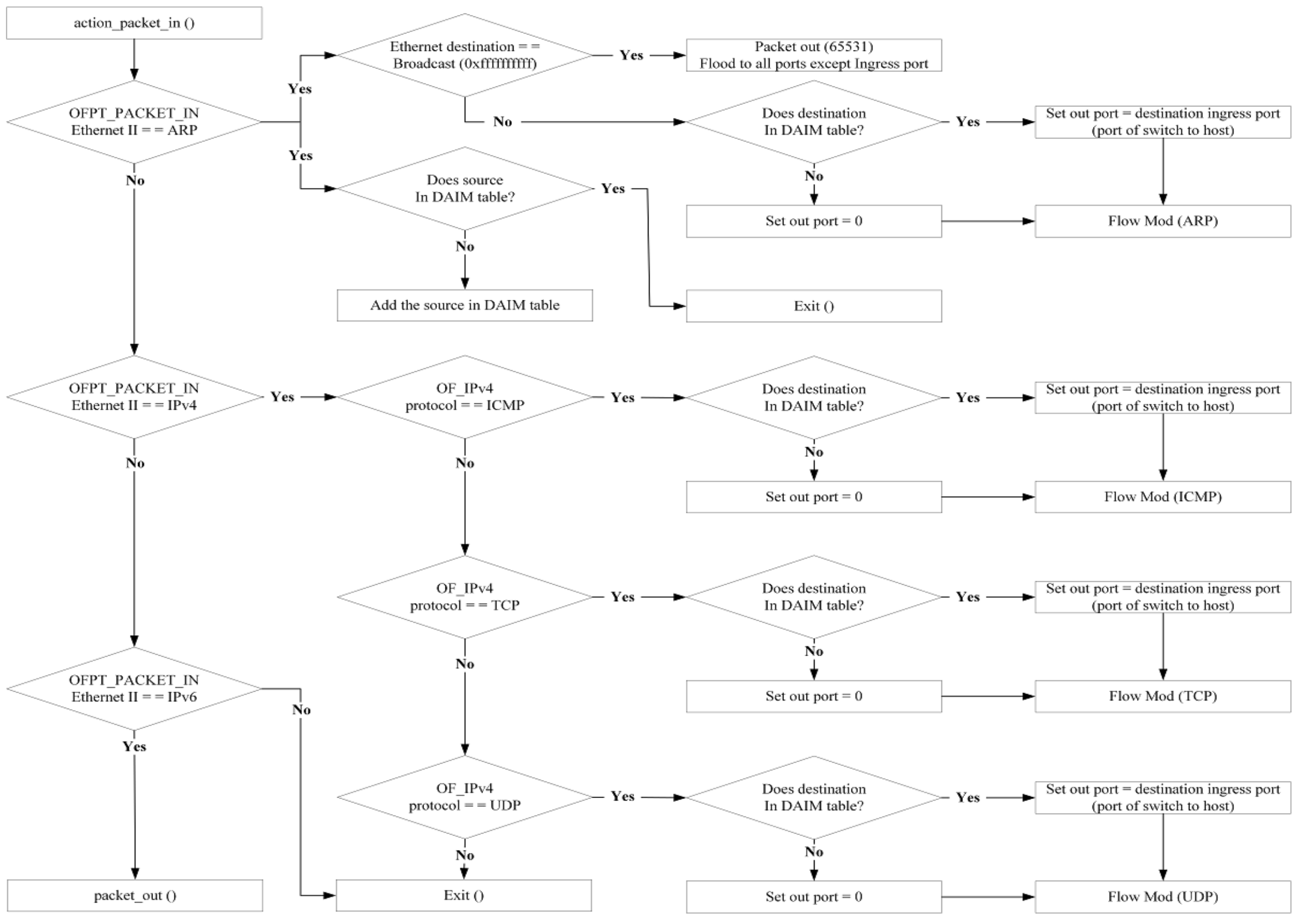

Fig. 7. Packet Flow in an OpenFlow Switch Controlled by DAIM

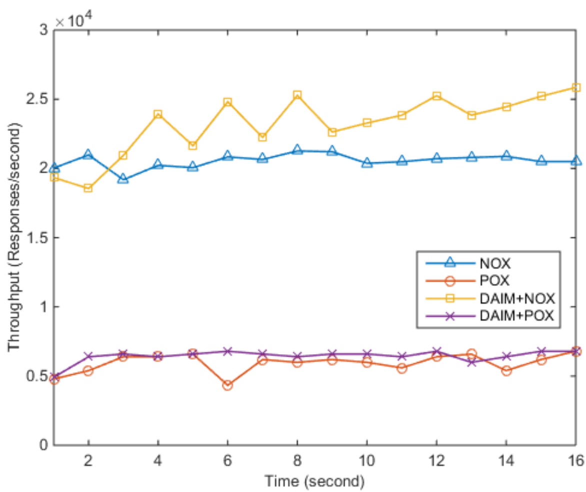

Fig. 8. Number of Flow Requests Handled per Second

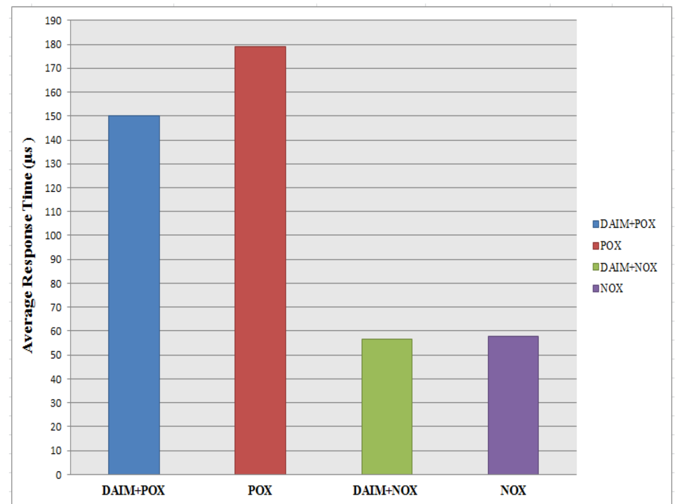

Fig. 9. Delay to Respond to Flow Requests

NOX with 22,357 responses per second. The lowest throughput was seen in the Python-based POX controller, which served 6,096 responses per second. Although DAIM runs slower than NOX-MT, it can be seen that the performance of DAIM outperforms the next best non-multi-threaded controller, NOX, by more than 7 times.

It can also be seen that the number of connected hosts in the network had an effect for two of the controllers, NOX-MT and DAIM. For example, NOX-MT's maximum throughput reduces from 292K to 239K responses per second where there are 10^7 hosts. Further, the performance of DAIM is seen to decrease when more than 10^5 hosts are connected. It is

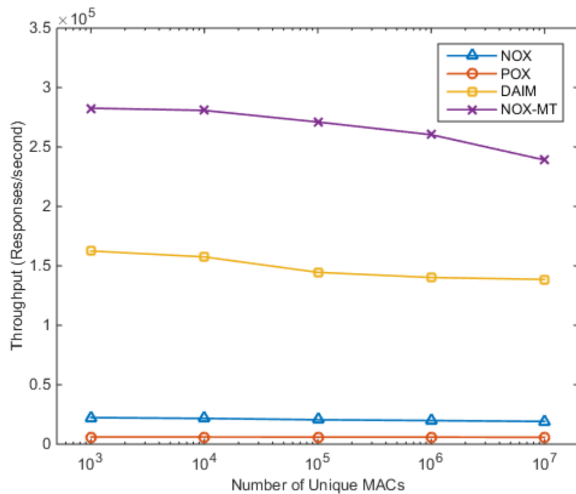

Fig. 10. Average Maximum Throughput Achieved with Different Numbers of MACs

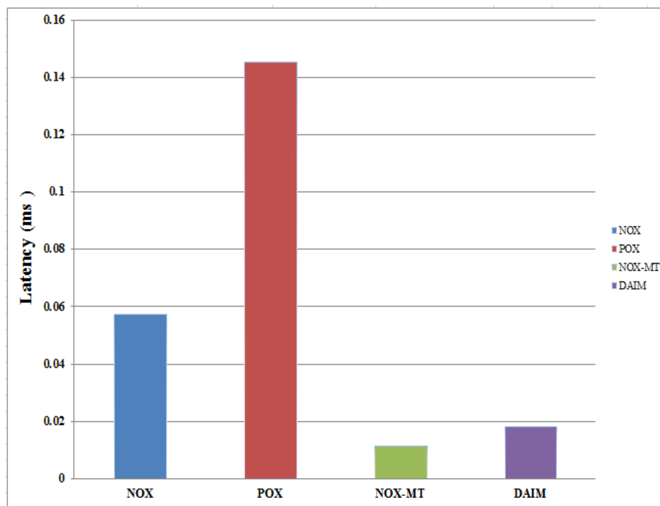

Fig. 11. Flow Setup Latency Comparison

thought that these effects are due to the implementation of the learning switch application, especially the implementation of the lookup table.

Latency: This test looked at the average latency seen in the different controllers when running a learning switch application. It should be noted that there are differences in the implementation of the learning switch in each of the different controllers.

To evaluate the latency of the controllers, the delay of processing flow requests was analysed with one switch, and 10^5 hosts. The results are shown in Figure 11. The lowest average latency was achieved by the NOX-MT and DAIM controllers, with 0.011 ms and 0.018 ms respectively. The average time it took the NOX controller to process each response was 0.057 ms, while the greatest average latency was seen in the POX controller: 0.145 ms. If the network has a high latency, packets take longer to be delivered, the probability of packet loss is increased, and overall network performance is reduced. As a result, POX is more suitable for prototyping than for enterprise deployment.

These results suggest that the DAIM controller can handle a high throughput, while keeping the latency relatively low.

VII. Conclusion

The DAIM controller was primarily developed to investigate the viability of a logically distributed control plane, achieved by integrating the DAIM controller in a distributed way throughout the network at the level of the switches. A performance analysis was conducted using the Cbench open benchmarking tool. Performance comparisons with pre-existing controllers were suggest that the DAIM controller can handle a high throughput, while keeping the latency relatively low. These findings are especially interesting given that a key feature of DAIM is that it is designed to enable the future development of autonomous local process flow management strategies.

VIII. Acknowledgment

This work is sponsored by the Centre for Real-Time Information Networks (CRIN) in the Faculty of Engineering & Information Technology at the University of Technology, Sydney (UTS), Australia

References

[1] O. N. Foundation, “Software-defined networking: The new norm for networks,” *ONF White Paper*, vol. 2, pp. 2–6, 2012.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: enabling innovation in campus networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[3] D. Jankowski and M. Amanowicz, “On efficiency of selected machine learning algorithms for intrusion detection in software defined networks,” *International Journal of Electronics and Telecommunications*, vol. 62, no. 3, pp. 247–252, 2016.

[4] I. Guis, “Enterprise data center networks,” *Open Networking Summit 2012*, 2012.

[5] L. R. Bays and D. S. Marcon, “Flow based load balancing: Optimizing web servers resource utilization,” *Journal of Applied Computing Research*, vol. 1, no. 2, pp. 76–83, 2011.

[6] S. Hassas Yeganeh and Y. Ganjali, “Kandoo: a framework for efficient and scalable offloading of control applications,” in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 19–24.

[7] A. Tootoonchian and Y. Ganjali, “Hyperflow: A distributed control plane for openflow,” in *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, 2010, pp. 3–3.

[8] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, “Onix: A distributed control platform for large-scale production networks.” in *OSDI*, vol. 10, 2010, pp. 1–6.

[9] A. Banjar, P. Pupatwibul, and R. Braun, “Daim: a mechanism to distribute control functions within openflow switches.” *JNW*, vol. 9, no. 1, pp. 1–9, 2014.

[10] P. Pupatwibul, A. Banjar, and R. Braun, “Using daim as a reactive interpreter for openflow networks to enable autonomic functionality,” *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 523–524, 2013.

[11] P. Pupatwibul, A. Banjar, A. Al Sabbagh, and R. Braun, “A comparative review: Accurate openflow simulation tools for prototyping.” *JNW*, vol. 10, no. 5, pp. 322–327, 2015.

[12] P. Pupatwibul, A. Banjar, A. A. Sabbagh, and R. Braun, “An intelligent model for distributed systems in next generation networks,” in *Advanced Methods and Applications in Computational Intelligence*. Springer, 2014, pp. 315–334.

[13] P. Pupatwibul, A. Banjar, A. Al Sabbagh, and R. Braun, “Developing an application based on openflow to enhance mobile ip networks,” in *IEEE Conference on Local Computer Networks*. IEEE, 2013.