

# A Fast Consensus Algorithm for Multiple Controllers in Software-Defined Networks

Chia-Chen Ho, Kuochen Wang, Yi-Huai Hsu

Department of Computer Science, National Chiao Tung University, Hsinchu 300, Taiwan

miniaoe.cs02g@nctu.edu.tw, {kwang, yhhsu}@cs.nctu.edu.tw

**Abstract**—Multiple SDN controllers architecture has been proposed to improve the scalability problem and to avoid a single point of failure. One major issue in the SDN multiple controllers architecture is how to reach a consistent network state among SDN controllers. In order to resolve the above issue and to make sure that all controllers have the same network state and the whole network becomes consistent, an efficient consensus mechanism to synchronize the control state of each controller is required. In view of this, we propose a *Fast Paxos-based Consensus* algorithm (FPC) which provides strong consistency. The proposed FPC uses a controller priority mechanism to guarantee a proposal must be elected in each round and no additional round is needed even more than two proposers get the same votes. Using Estinet, simulation results have shown that the proposed FPC has lower average consensus time (35.3% lower) than Raft. With a low consensus time, the proposed FPC can improve the data store access performance (26.0% faster at retrieving data and 59.7% faster at storing data via the REST API comparing with the Raft). Therefore, the proposed FPC is feasible for multiple SDN controller networks.

**Keywords**—Consensus algorithm, consistency, multiple controllers, SDN.

## I. INTRODUCTION

The Software Defined Network (SDN) has been proposed as a promising network technology to improve network flexibility and simplify network management. The majority of current SDN architecture uses a single controller. However, the single controller architecture leads to some problems such as a single point of failure, scalability and security issues. Existing studies proposed the multiple SDN controllers architecture to address the above issues. When implementing a multiple SDN controllers architecture, how to synchronize the network state information among controllers, known as the controller consensus problem, is a critical problem. With a low consensus time, data store access performance can be improved. In order to synchronize the network information of controllers, we should select a proper consistency model. Strong consistency [1] and eventual consistency [1] are two consistency models commonly used in distributed file systems. The strong consistency model has longer data store access time but can assure that the controller have the latest updated network information, while the eventual consistency model has shorter data store access time but the controller may not get the latest updated network information. In SDNs, latest network information is needed for most applications. For example, an SDN routing application needs the latest topology information to generate a correct routing path.

Therefore, we prefer to build a multiple SDN controllers architecture with a strong consistency model. The objective of this paper is to resolve the controller consensus problem and to further improve data store access performance.

In this paper, we propose a *Fast Paxos-based Consensus* (FPC) algorithm based on a strong consistency model, Paxos [2], to handle the controller consensus problem. The proposed FPC has stable consensus time and can improve the data store access time performance. This paper is organized as follows. Section II describes some existing consensus mechanisms used in major multiple SDN controllers. Section III details the proposed FPC algorithm. Experiment results are discussed in Section IV. Finally, we conclude this paper in Section V.

## II. RELATED WORK

In this section, we describe related work on consensus mechanisms for multiple SDN controllers. Many multiple SDN controllers architectures have been proposed to achieve scalability and reliability. Each architecture has its own mechanism to synchronize the network state among all controllers. We classify these consensus mechanisms according to their consistency models, which include eventual consistency and strong consistency.

### A. Eventual consistency

HyperFlow [3], an event-based control plane for OpenFlow, is logically centralized but physically distributed. HyperFlow passively synchronizes the network state among controllers through a publish/subscribe system [4] based on the WheelFS distributed file system [5]. HyperFlow selectively publishes the events that change the network state of the controllers, such as a switch sends a *PacketIn* message to a controller. Then other controllers replay the published events to update the network state. By doing so, it can achieve a consistent network-wide view among controllers. Onix [6] is a multiple SDN controllers architecture that provides a control application with a set of general APIs to facilitate access to the network state. In Onix, the controller stores network information in key value pairs by utilizing the Network Information Base (NIB), which is the core element of the Onix model. The Onix synchronizes the network state by reading and writing to the NIB, and the Onix provides scalability and resilience by replicating and distributing the NIB between multiple NIB instances. Once there is a change of an NIB on one Onix node, the change will be propagated to other NIBs to maintain the consistency of the network.

**Table 1.** Comparison of related works on multiple controllers consensus mechanisms.

| Property Design           | Mechanism         | Contents                                                | Controller Consensus                         | Consistency |
|---------------------------|-------------------|---------------------------------------------------------|----------------------------------------------|-------------|
| HyperFlow [3]             | DFS               | OpenFlow events                                         | Publish/Subscribe (Polling DFS, leader free) | Eventual    |
| Onix [6]                  | DHT, DFS          | NIB                                                     | Onix API (leader free)                       | Eventual    |
| GRACE [7]                 | DHT, Grace layer  | Control State                                           | Byzantine Paxos (leader free)                | Strong      |
| DISCO [10]                | Messenger, Agents | Link status, Controller status, Intra-domain flow table | Publish/Subscribe (AMQP, leader free)        | Strong      |
| ODL Clustering [13]       | DHT, Akka         | Control State                                           | Raft                                         | Strong      |
| MBL Clustering (proposed) | DHT, Akka         | Control State                                           | FPC (proposed, leader free)                  | Strong      |

### B. Strong consistency

In [7], it introduced two core features of its design, the reconfiguration primitive of the data plane and the state reuse primitive of the service control state. The reconfiguration primitive is used to update the forwarding state of the data plane and all controllers consent on a global network state through the Byzantine Paxos algorithm [8]. The state reuse primitive allows applications to access the network state. The network information of a service is stored and organized using the DHT. These two primitives are implemented in the GRACE layer which sits between the OpenFlow protocol stack and the distributed services of the NOX controller [9]. DISCO [10] is a multiple SDN controllers architecture which can deal with the wide area network. The DISCO uses a messenger module and four types of agents, which are monitoring, reachability, connectivity and reservation, to maintain controller consistency. The messenger module is based on the AMQP (advanced message queuing protocol) [11] and its work is to discover neighboring controllers and maintain a distributed publish/subscribe communication channel. Different agents use this channel to the messenger to exchange network-wide information with other controllers. Each agent publishes and handles messages according to its purpose and works with the messenger to ensure the consistency of the network. OpenDaylight (ODL) [12] provides a multiple SDN controllers architecture called ODL Clustering [13]. ODL Clustering is a mechanism that enables multiple processes and programs to work together as one entity. Each controller has its own data store called shard and the data store it uses is Infinispan replicated caches [14]. Akka [15] is a toolkit used in ODL Clustering and is responsible for communication and notification among controllers. In the default clustering scheme, switches connect to all controllers, and these controllers coordinate

among themselves in order to choose a master controller. The ODL uses the Raft algorithm [16] to reach controllers consistency. The Raft consensus algorithm periodically elects a controller as a leader controller, and all data changes will be sent to the leader controller to handle the update. Comparison of related works on multiple controllers consensus mechanisms and the proposed FPC are summarized in Table 1.

### III. PROPOSED FAST PAXOS-BASED CONTROLLER CONSENSUS ALGORITHM FOR MULTIPLE SDN CONTROLLERS

We propose a *Fast Paxos-based Controller Consensus Algorithm for Multiple Controllers (FPC)* in SDNs to reach a consistent network state among multiple SDN controllers. The proposed consensus algorithm is based on a famous strong consistency model, Paxos. Paxos is a family of protocols for solving consensus in a network of unreliable processors, used by Google Chubby [17] and Microsoft Autopilot [19]. Paxos can guarantee strong consistency in the distributed file system. However, Paxos only proposed the concept of leader election, failure detection, and log management, but the details of these concepts are not defined in the Paxos specification. With the proposed FPC, we can reduce the complexity in developing and implementing Paxos. The details of the proposed FPC are described as follows.

The proposed consensus algorithm has three roles, *Listener*, *Proposer*, and *Chairman*. When a controller runs, its initial role is *Listener*. A *Listener* will switch to a *Proposer* only if it receives a new request (e.g., a data change). A *Proposer* first sends a proposal to all controllers and then waits for proposal reply. If the *Proposer* receives acceptance votes from a majority of controllers, it will switch to a *Chairman*. Otherwise, it switches back to a *Listener*. The *Chairman* will update the request and send update information to the other controllers. Once all controllers have done the update, the *Chairman* will switch to a *Listener* and ends the consensus process. Figure 1 shows the role change state machine of a controller. We monitor all controllers with a controller state list, which records all controllers' states, and we assume that all messages will reach the destination controllers eventually. The proposed FPC is composed with four phases, *Propose*, *Accept*, *Update*, and *Adjust*. A controller keeps a priority table to record its current controller priority, and the initial value is the controller's id. We define that the lower the id, the higher the priority. In the following, we describe the four phases. Table 2 shows notations that are used in the proposed FPC and their definitions.

In the *Propose* phase, the controller that receives a new request (a *Proposer*) will send a proposal message *MPropose* to the other controllers. If the *Proposer* receives acceptance votes from more than half of the controllers, it will switch to the role of *Chairman* and go to the *Update* phase. But if the *Proposer* receives a message *MError* to notify the *Proposer* that its  $T_jRN$  is incorrect from more than half of the controllers, it'll need to give up this round and go to the *Adjust* phase to make sure that it is in the newest state as the other controllers. If the *Proposer* doesn't receive acceptance

messages from a majority of controllers and receive all proposal reply messages *MReply*, it'll check whether it is the *Proposer* with the highest priority or not. If it is not the one with the highest priority, it would forward all acceptance votes it has to the *Proposer* with the highest priority and switch back to the role of *Listener*. In the *Accept* phase, for any controller that receives a proposal message *MPropose*, if it finds the *Proposer*'s round number ( $T_jRN$ ) is different from its transaction round number, it will send an *MError* message back. If it hasn't accept any proposal of that transaction type and it is not a *Proposer* of that type too, it accepts the proposal. Otherwise, it rejects the proposal.

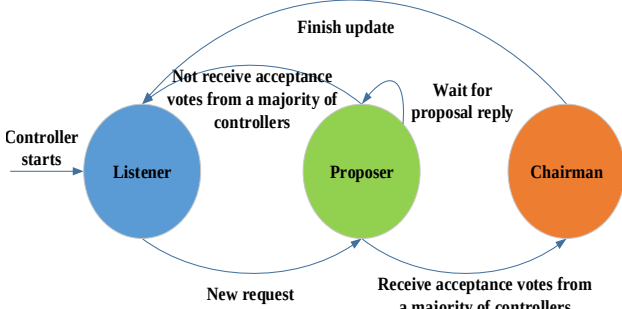


Figure 1. The role change state machine of a controller.

Table 2. Notations and definitions.

| Notation           | Definition                                                                                                     |
|--------------------|----------------------------------------------------------------------------------------------------------------|
| $C_i$              | $i^{th}$ controller                                                                                            |
| $T_j$              | Transaction type $j$ , where $j$ can be topology or link capacity                                              |
| $T_jLPriority$     | Current lowest priority of $T_j$                                                                               |
| $T_jCPriority$     | Controller's current priority of $T_j$                                                                         |
| $T_jPromise$       | Does the controller promise to accept a $T_j$ proposal or not                                                  |
| $T_jRN$            | Current transaction round number of $T_j$                                                                      |
| <i>MPropose</i>    | Proposal message, containing the <i>Proposer</i> 's $C_i$ , $T_jRN$ and $T_jCPriority$                         |
| <i>MReply</i>      | Proposal reply message, containing the sender's votes, accepted $C_i$ and accepted $C_i$ 's $T_jCPriority$     |
| <i>MError</i>      | A message to notify the <i>Proposer</i> that the <i>Proposer</i> 's $T_jRN$ is incorrect                       |
| <i>MReady</i>      | A message to notify a controller to be ready to do update, containing the <i>Chairman</i> 's $T_jRN$ and $C_i$ |
| <i>MRReply</i>     | A message to notify the <i>Chairman</i> that the controller is ready to do update                              |
| <i>MAdjust</i>     | A message to notify the <i>Chairman</i> that the <i>Listener</i> needs an update log to adjust itself          |
| <i>MAREply</i>     | A reply for <i>MAdjust</i> , containing the <i>Chairman</i> 's update log                                      |
| <i>MUpdateDone</i> | A message to notify the <i>Chairman</i> that it has done the update                                            |
| <i>MRoundEnd</i>   | A message to notify all controllers that this is the end of a update round                                     |
| <i>MUpdate</i>     | A message containing the data that needs to be updated                                                         |

In the *Update* phase, the *Chairman* first sends an *MReady* message to the other controllers to notify them to be ready to do update. Once the *Chairman* received an *MRReply* reply from the other controllers, it will start to update itself

and then send update information to the other controllers for doing update. If it receives an *MAdjust* message from a controller to ask for an update log, it'll send an *MAREply* message contains the update log to that controller. During the update time, the *Proposer* will monitor the other controllers to make sure that they have done the update. For the other controllers, it will periodically check the controller state list to see if the *Chairman* failed. Once they received an *MReady* message from the *Chairman*, it will check  $T_jRN$  to see if it's in the newest state. If its transaction round number is different from that of the *Chairman*, it'll send an *MAdjust* message to the *Chairman* and update itself to the newest state first then send an *MReady* message to the *Chairman*. Otherwise, it will reply the *Chairman* with an *MReady* message directly and wait for an update message. As soon as a controller receives the update message, it'll do the update. After the update is done, the controller will send an *MUpdateDone* message to the *Chairman*. Once the *Chairman* received all *MUpdateDone* messages, it will send *MRoundEnd* messages to the other controllers to announce that this is the end of update round. The *Listener* will reset  $T_iPromise$  and  $T_jACP$ , and update  $T_iRN$  and  $T_jPriority$  after received an *MRoundEnd* message from the *Chairman*. The *Chairman* will update  $T_iRN$ ,  $T_jCPriority$  and  $T_jPriority$ , then switch to the role of *Listener*. The equations (1), (2) and (3) are the updates of  $T_iRN$ ,  $T_jPriority$  and  $T_jCPriority$ , respectively.

$$T_iRN = T_iRN + 1 \quad (1)$$

$$T_jLPriority = T_jLPriority + 1 \quad (2)$$

$$T_jCPriority = T_jLPriority \quad (3)$$

The *Adjust* phase is used for a new joined controller. The new joined controller will broadcast a message to get an update log from other controllers, and the other controllers will update  $T_jCPriority$  according to equation (3) and send its  $T_iRN$  to the new joined controller. Once the new joined controller receives replies from the other controllers, it will send an adjust message to the controller with the highest  $T_iRN$ . The controller who received the adjust message will send its update log to the new joined controller. The new joined controller then updates itself according to the newest log.

#### IV. EXPERIMENT RESULTS

##### A. Simulation setup

Table 3 shows our simulation parameters. We used the ODL OpenFlow controller [12] as the control plane, and we used an EstiNet OpenFlow network emulator [18] to implement the data plane. The environment of the virtual machine (VM) is Ubuntu 14.04 LTS. The hardware is IBM System 3650M3 (Esxi 5.5). We evaluated the performance of the proposed design and the original ODL Clustering on 3-node clustering. The Raft is a consensus algorithm and is used in ODL Clustering. The original Raft algorithm periodically elects a controller as the leader controller, which

is the master controller in the ODL Clustering. As for communications between controllers, the OpenDaylight Clustering uses Akka to handle it. Akka is a toolkit and runtime for simplifying the construction of concurrent and distributed resilient message-driven applications on the JVM (Java virtual machine) [15].

**Table 3.** Simulation parameters.

| Parameter             | Value                                  |
|-----------------------|----------------------------------------|
| Control plane         | ODL OpenFlow controller [14]           |
| Data plane            | EstiNet OpenFlow network emulator [22] |
| Environment           | Ubuntu 14.04 LTS                       |
| Hardware              | IBM System 3650M3 (Esxi 5.5)           |
| Number of controllers | 3 [6][7][10]                           |

A 3-node ODL Clustering architecture is shown in Figure 2. We implemented the proposed FPC on the ODL Clustering project [15]. Toaster [20] is a test model for testing ODL MD-SAL (Model-Driven Service Abstraction Layer) functionality. We used the toaster model and modified the data values of the toaster to form a toaster REST API request. We executed 1000 requests for storing toaster data and 1000 requests for retrieving toaster data on each controller. We compare the proposed FPC algorithm and the Raft consensus algorithm used in ODL Clustering, in terms of consensus time, distribution of normalized consensus time and toaster REST API data access time. The consensus time is the time elapsed from a transaction created to the transaction committed. We used the distribution of normalized consensus time to reflect the controller loading issue. The ODL Clustering uses Akka to handle all request messages, and has a message queue for each controller. So when a controller's loading becomes higher, request messages may be queued. As a result, it affects and prolongs the consensus time, so the consensus time may become high under high load. The toaster REST API data access time is the time when a request is called by REST API to the time the result returns, which shows how consensus time affects data store access time.

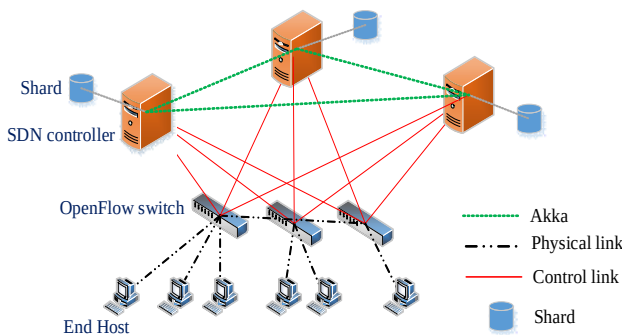


**Figure 2.** A 3-node ODL SDN Clustering architecture.

## B. Experimental results

Figure 3 shows the proposed FPC has 35.3% lower average consensus time than the Raft. The main reason is that no matter where a request comes from, in the proposed FPC, the controller can directly handle the request while the Raft

may need to send the request to the leader controller. In addition, when the leader controller's loading becomes high, it cannot handle all queued messages in time. Thus, the consensus time may go very high. Figure 4 shows that the consensus time of the proposed FPC is more stable and the overall consensus time is lower, compared to that of the Raft. With 95% confidence level, the Raft has a confidence interval from 8 to 67 milliseconds and the variance is 20.105 when the request is generated by a follower controller. When the request is generated by the leader controller, with 95% confidence level, the Raft has a confidence interval from 6 to 14 milliseconds and the variance is 3.548. In contrast, the proposed FPC has a confidence interval from 6 to 28 milliseconds and the variance is 7.331.

Having a low and stable consensus time can improve the performance of some REST APIs which needs to access data stores and can also decrease the flow setup time. Table 4 shows the toaster REST API data access time after executing 1000 toaster requests on each controller with 3 nodes clustering. The proposed FPC can improve the data store access performance (26.0% faster at retrieving data and 59.7% faster at storing data via toaster REST API, compared with the Raft).

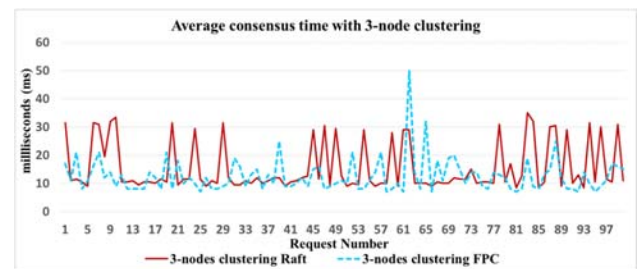


**Figure 3.** Average consensus time of 3-node ODL clustering.

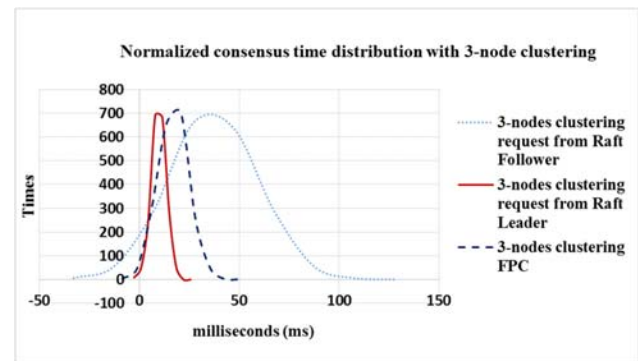


**Figure 4.** Normalized consensus time distribution with 3-node ODL Clustering.

**Table 4.** Toaster REST API average data access time.

| Scheme                                  | Read/Write |            |
|-----------------------------------------|------------|------------|
|                                         | Read Data  | Write Data |
| Raft (Request from leader controller)   | 5.415ms    | 11.691ms   |
| Raft (Request from follower controller) | 10.323ms   | 60.513ms   |
| Raft                                    | 8.687ms    | 44.239ms   |
| FPC (proposed)                          | 6.425ms    | 17.814ms   |

## V. CONCLUSIONS

In this paper, we have presented a Fast Paxos-based Consensus (FPC) algorithm to handle the consensus issue of multiple SDN controllers. The proposed FPC simplifies the original Paxos protocols and can alleviate the complexity in developing and implementing Paxos. The proposed FPC contains three roles, *Listener*, *Proposer* and *Chairman*. When receiving a request, a *Listener* will become a *Proposer* and it will start a round and ask the other controllers whether they accept this request or not. If the *Proposer* receives acceptance votes from a majority of the controllers, it can handle the request and the other controllers will do the update. In addition, each controller has a unique controller priority and an aging mechanism is used to prevent low priority controllers from starvation. When no *Proposer* receives acceptance votes from a majority of controllers, each *Proposer* will check its controller priority and vote the *Proposer* with the highest priority. With a unique controller priority, the proposed FPC can guarantee that a proposal is elected in each round and no additional round is needed even more than two *Proposers* got the same votes. Simulation results have shown that the proposed FPC has lower average consensus time (35.3% lower) than the Raft. With a low consensus time, the proposed FPC can improve the data store access performance (26.0% faster at retrieving data and 59.7% faster at storing data via the toaster REST API, compared with the Raft).

## ACKNOWLEDGMENTS

The support by the Ministry of Science and Technology under Grant Most 103-2622-E-009-012 and by the EstiNet Technologies under Contract 103C138 is gratefully acknowledged.

## REFERENCES

- [1] Consistency model [Online]. Available: [https://en.wikipedia.org/wiki/Consistency\\_model](https://en.wikipedia.org/wiki/Consistency_model).
- [2] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133-169, May 1998.
- [3] A. Tootoonchian and Y. Ganjali, "HyperFlow: a distributed control plane for OpenFlow," in *Proc. ACM Internet Network Management Conference on Research on Enterprise Networking*, April 2010.
- [4] Publish/subscribe [Online]. Available: [https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe\\_pattern](https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern).
- [5] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek and R. Morris, "Flexible, wide-area storage for distributed systems with WheelFS," in *Proc. USENIX Symposium on Networked Systems Design and Implementation*, vol. 9, pp. 43-58, April 2009.
- [6] T. Koponen, M. Casado, N. Gude, J. Stribling, and L. Poutievski, "Onix: A distributed control platform for large-scale production networks," in *Proc. USENIX Symposium on Networked Systems Design and Implementation*, pp. 1-6, Oct. 2010.
- [7] Zhou et al., "Programming network via distributed control in software-defined networks," in *Proc. IEEE International Conference on Communications*, pp. 3051-3057, June 2014.
- [8] L. Lamport, "Byzantizing paxos by refinement," *Distributed Computing*, Springer, pp. 211-224, 2011.
- [9] 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*, 38(3), pp. 105-110, July 2008.
- [10] K. Phemius, M. Bouet, and J. Leguay, "Disco: Distributed multi-domain SDN controllers," in *Proc. IEEE Network Operations and Management Symposium*, pp. 1-4, May 2014.
- [11] *Advanced Message Queuing Protocol (AMQP)* [Online]. Available: <http://www.amqp.org>.
- [12] *OpenDaylight Linux Foundation Collaborative Projects* [Online]. Available: <http://www.opendaylight.org/>.
- [13] *OpenDaylight SDN Controller Platform (OSCP): Clustering* [Online]. Available: [https://wiki.opendaylight.org/view/OpenDaylight\\_SDN\\_Controller\\_Platform\\_%28OSCP%29:Clustering](https://wiki.opendaylight.org/view/OpenDaylight_SDN_Controller_Platform_%28OSCP%29:Clustering).
- [14] *Infinispan* [Online]. Available: <http://infinispan.org/>.
- [15] *Akka* [Online]. Available: [akka.io](http://akka.io).
- [16] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annual Technical Conference*, pp. 305-320, June 2014.
- [17] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proc. of the 7th Symposium on Operating Systems Design and Implementation*, USENIX Association, pp. 335-350, Nov. 2006.
- [18] *EstiNet OpenFlow network emulator* [Online]. Available: <http://www.estinet.com>.
- [19] M. Isard, "Autopilot: Automatic data center management," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 2, pp. 60-67, 2007.
- [20] OpenDaylight Toaster [Online]. Available: [https://wiki.opendaylight.org/view/OpenDaylight\\_Controller:MD-SAL:Toaster\\_Step-By-Step](https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Toaster_Step-By-Step).



**Chia-Chen Ho** received the B.S. degree in the Department of Computer Science from the National Cheng Kung University, Tainan, Taiwan, in 2013. She received the M.S. degree in the Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan, in 2015. Her research interests include software defined networking and cloud computing.



**Kuochen Wang** received the B.S. degree in control engineering from the National Chiao Tung University, Taiwan, in 1978, and the M.S. and Ph.D. degrees in electrical engineering from the University of Arizona in 1986 and 1991, respectively. He is currently a Professor and the Chair of the Department of Computer Science, National Chiao Tung University. He was a Director of the Institute of Computer Science and Engineering/Institute of Network Engineering, National Chiao Tung University from August 2009 to July 2011. He was an Acting/Deputy Director of the Computer and Network Center at this university from June 2007 to July 2009. He was a Visiting Scholar in the Department of Electrical Engineering, University of Washington from July 2001 to February 2002. From 1980 to 1984, he was a Senior Engineer at the Directorate General of Telecommunications in Taiwan. He served in the army as a second lieutenant communication platoon leader from 1978 to 1980. His research interests include cloud computing and software defined networking, internet of things and big data analytics, energy-aware mobile computing and networking, and dependable computing and networks.



**Yi-Huai Hsu** received the B.S. degree in Computer and Information Science from the National Taichung University, Taichung, Taiwan, in 2008 and the M.S. degree in Computer Science from the National Chiao Tung University, Taiwan, in 2010. He is currently a Ph.D. candidate in the Department of Computer Science, National Chiao Tung University. His research interests include wireless (ad hoc/sensor/VANET) networks, cloud computing, and LTE-A.