Bootstrapping Software Defined Network for Flexible and Dynamic Control Plane Management

Prithviraj Patil, Aniruddha Gokhale, Akram Hakiri

Bootstrapping Software Defined Network for Flexible and Dynamic Control Plane Management

Prithviraj Patil*, Aniruddha Gokhale* and Akram Hakiri†
*ISIS, Dept of EECS, Vanderbilt University, Nashville, TN, USA.
†CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
Email: {prithviraj.p.patil,a.gokhale}@vanderbilt.edu, hakiri@laas.fr

***Abstract*—To improve reliability and performance of Software Defined Networking (SDN) architectures, a number of recent efforts have proposed a logically centralized but physically distributed controller design that overcomes the bottleneck introduced by a single physical controller. Despite these advances, two key problems still persist. First, the task of controlling the host network and the task of controlling the control-plane network remain tightly intertwined, which incurs unwanted complexity in the controller design. Second, the task of deploying the distributed controllers continues to be performed in a manual and static way. To address these two problems, this paper presents a novel approach called InitSDN to bootstrapping the distributed software defined network architecture and deploying the distributed controllers. InitSDN makes the SDN control plane design less complex, makes coordination among controllers flexible, provides additional reliability to the distributed control plane.**

I. Introduction:

A. Software Defined Networking and Emerging Challenges

SDN architecture envisions a centralized control plane, which may result in adverse consequences to the reliability and performance [**?**]. Recent efforts have proposed a logically centralized but physically distributed control plane [**?**]. The distributed control plane is more responsive to handle network events because the controllers tend to be closer to the events than the centralized architecture. However, these solutions incur a different set of complexities for developing and managing the controllers. One key limitation of these approaches is that they club task of controlling the host network and task of managing the distributed control plane together. Hence, the developer of a distributed controller now has to take care of all the concerns that arise out of distributed nature of the system including controller synchronization, controller replication, controller logic partitioning and controller placement [**?**], [**?**]. All the above issues are orthogonal to the fundamental controller functionality. However current distributed control plane architecture forces controller developer to invest energy into addressing these issues which complicates the controller design and management and makes control-plane inflexible.

B. Proposed Solution and Contributions

To address these problems, we propose a solution called *InitSDN,* which is based on a bootstrapping mechanism that helps to decouple the orthogonal distributed systems concerns from the primary issues related to the controller. InitSDN is designed to make SDN more flexible, reliable, fault-tolerant without adding complexity to the controllers.

InitSDN divides a single physical network substrate into two slices: a *dataslice* for controlling the hosts that run user applications and a *controlslice* for controlling the controllers. Based on the configuration or strategy defined by a network operator, InitSDN allocates the right number of hosts between these two slices,[1] selects an initial topology for the *controlslice*, deploys required controllers in the *controlslice*, sets the coordination mechanism among the controllers, maps the switches in the *dataslice* to distributed controllers, and kick-starts the operation of the real/actual SDN. Over the course of the SDN operation, InitSDN can increase or decrease the size of slices dynamically, change the topology of the *controlslice*, change the coordination mechanism among the controllers (e.g. use Zookeeper or Chubby, etc) to adapt to network topology changes or to dynamic network loads or simply as part of an upgrade.

In the context of our InitSDN ideas, we make the following three contributions in this paper:

- We propose and describe the architecture of the InitSDN controller used for bootstrapping a real SDN network,
- We describe the implementation details of the InitSDN controller.
- We qualitatively evaluate the benefits of our approach in terms of separation of concerns, reduced complexity of the SDN controller, increased reliability and better management of control-plane using various motivating use cases.

II. Problem Definition

In this section, we provide a detailed motivation for a initSDN controller.

A. Control Plane Message Types

We categorize messages that are being exchanged in the SDN in three different categories as described below:

1) *Control messages:* These are the messages that are used to control the communication between the hosts. It includes various OpenFlow messages like

[1]In a shared or in-band control network, which is our focus, the controller logic must reside on some host of that network and hence some hosts will be used for hosting the controller logic while others will be used for application logic.

OFPT_FLOW-MOD, OFPT_FLOW_REMOVED, OFPT_PACEKT_IN, OFPT_PACKET_OUT, OFPT_GET_CONFIG_REQUEST, OFPT_SET_CONFIG, etc. These messages flow between controller-switch pairs.

2) *Data messsages:* These are normal data packets sent/received by hosts. These messages normally flow between switch-host or switch-switch pairs.
3) *Meta-control messages:* We define meta-control messages as those messages that are used to control the communication between SDN infrastructure entities, i.e. controllers, switches. It includes all the messages that are required for controller-switch connection setup, connection tear-down, controller-migration, switch-migration, host-migration, network discovery and topology services, controller logic synchronization or backup, etc. These messages flow between controller-switch, controller-controller and switch-switch pairs. It can includes OpenFlow messages like OFPT_FLOW-MOD, OFPT_FLOW_REMOVED, OFPT_PACEKT_IN, OFPT_PACKET_OUT, OFPT_GET_CONFIG_REQUEST, OFPT_SET_CONFIG, etc. Also in addition to above OpenFlow messages, it may include other non-OpenFlow non-standardized messages and different solutions may implement them in their own proprietary manner.

B. Limitations of Existing Control Plane

A number of prior studies have proposed designs for a distributed, scalable, and fault tolerant controller architecture in the SDN [**?**], [**?**], [**?**]. A key commonality across these approaches is to add a connection management module in the controller alongside the Openflow module. This module is responsible for tasks like leader election, synchronization, participation in switch migration, managing backups, state consistency, etc.

There are two basic problems with such distributed control plane design. First, in such architectures, the data messages flow in the SDN network but control messages flow in the non-SDN legacy network. This occurs because currently, control messages need to be exchanged to set up the SDN first. Then only after SDN is setup (i.e. switches are configured with correct controller references and flow-rules), data messages can be exchanged. Hence control messages are thought to be flowing in the pre-SDN (or non-SDN or legacy network).

Secondly, in these architectures, the control and meta-control messages are clubbed together, i.e. they originate from the same controller. This forces the controllers to handle many of the distributed system complexities, such as handling partitioning, placement, consensus, synchronization, coordination, which complicates the design of the controller and violates many of the software engineering principles resulting in code that is hard to maintain and evolve.

III. Design and Implementation of InitSDN

A. InitSDN Architecture

We now present the architecture and implementation details of the InitSDN approach.

Figure 1 shows the architecture of InitSDN. It works in the legacy network (i.e., non SDN) that uses the TCP/IP protocol. InitSDN has a modular structure with various modules as follows:

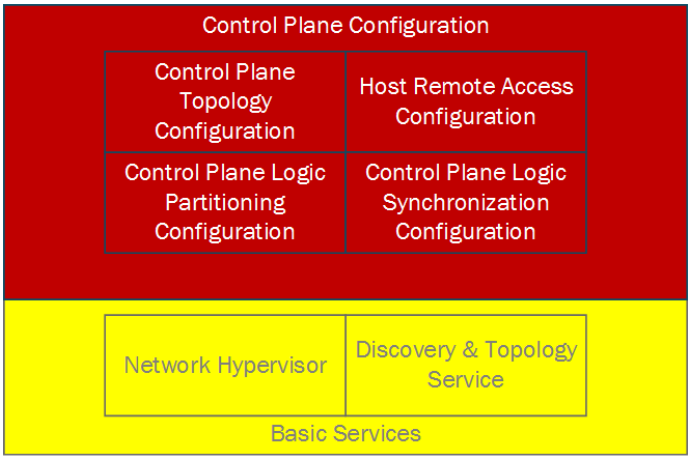

Fig. 1. **InitSDN modular architecture**

1) *Network discovery & topology service:* This is the basic module of the InitSDN. It discovers the switches and hosts in the network. It then creates the model of the network topology using specialized packets. It sends LLDP (Link Layer Discovery Protocol) packets to switches, parses the reply messages and builds the topology model.
2) *Network Hypervisor:* This module provides access to the existing network hypervisors. A network hypervisor is used to slice the network into control and data slice. Currently we have used Flowvisor [**?**]. This module is also used to create virtual switches for multi-tenant network applications. For this, currently we use OpenVirtex [**?**]. However, our design can accommodate other network hypervisors.
3) *Control-plane topology:* This module allows the network operator to specify the initial control plane topology. By default, InitSDN uses the basic topology with one centralized controller and one backup controller. Network operators however, can provide their own control plane topology as described below. This module then slices the network into two slices using information from the previous two modules (i.e., network hypervisor configuration and discovery & topology service).
4) *Control-plane partitioning:* This module is used to slice the control plane logic. This requires the controller to expose an API to perform this action. These APIs currently are controller-specific. In our present implementation, we have used a modified POX controller. For example,

Pyretic [**?**] has a modified POX client, which allows us to specify the flows to be controlled by the POX controller using a command line argument when starting the controller.

5) *Control-plane synchronization:* This module is used to specify the synchronization mechanism to be used in the control plane, e.g., how to synchronize the backup controller. Currently with the modified POX controller, we use Apache Zookeeper for synchronization. The modified POX controller writes its state (e.g. topology, counter etc) to a file. This file then gets synchronized across the control plane. This module allows an operator to use any other synchronization mechanism, e.g., Vagrant Serf, Google Chubby, etc.
6) *Host Remote Access:* Since InitSDN installs controllers on the hosts, it needs access to do so on those hosts. This module provides a way to configure such access. At present this module uses a combination of SSH and SCP through the Python command line tool Fabric [**?**]. However, based on the host access policy, the network operator can use any other tool.

B. InitSDN in Action

Now we describe the steps involved in the booting of a legacy network into a flexible, dynamic and fault tolerant SDN network using InitSDN.

1) *Initial Setup:* We assume a network substrate which uses a legacy network with Openflow-enabled switches. InitSDN has remote access to all the hosts that are supposed to host the control plane. The chosen SDN controller exposes the API to configure the partitioning and synchronization strategy.
2) InitSDN is started on one of the hosts in this network substrate and is connected to all (top-level) main switches statically.
3) An InitSDN network application will then configure the InitSDN controller. This InitSDN application contains configuration information of all the InitSDN control-plane modules shown in Figure 1 and also described in the previous Subsection III-A.
4) InitSDN will build a model of the topology of the network using the discovery and topology module. The topology contains all the hosts, switches and links present in the network. It will also contain link properties and switch configurations like the ones supported in the OpenFlow version.
5) InitSDN then builds the control-plane topology based on the configuration provided by the network operator and network topology model from the previous step.
6) Using the network hypervisor (e.g. flowvisor), InitSDN will slice the network into two slices namely data-slice and control-slice. The number of hosts in both the slices and their topology is determined by the control-plane topology from the previous step.
7) InitSDN then remotely installs the controller in all the hosts in the control plane.
8) InitSDN configures the controllers in the control plane as per the control plane partitioning strategy provided by the network operator, e.g., controller C1 handles only secure flows while controller C2 handles only non-secure flows, etc.
9) InitSDN configures the synchronization strategy in the control plane as the controllers need to share the local topology changes with the other (non-local or remote) controllers, e.g., backup controllers need to be synchronized with the respective primary controller, etc.
10) InitSDN then installs the default flow-rules in the switches so that in case of control plane failure, switch will notify InitSDN. This adds an additional level of reliability to the SDN control plane.
11) InitSDN then configures all the switches with one or more controllers from the control-slice.
12) At this point, SDN is considered to be booted as per the configuration provided by the network operator and InitSDN is out of the picture.

C. Implementation Details for Initial Prototype

The following tools and technologies were used to realize InitSDN and evaluate its properties.

1) *Network Emulation:* Mininet [**?**].
2) *Switch:* OpenVswitch and Openflow's Reference Switch (ofdatapath) [**?**].
3) *Controller:* Openflow's Reference Controller [**?**], Apache Floodlight, Stanford University's Pox and Ryu.
4) *Host:* Docker Containers and VirtualBox VMs.
5) *Network Virtualization:* Flowvisor [**?**], OpenVirtex [**?**].
6) *Network Topologies:* Real network topologies (built using traceroute) obtained from Stanford University [**?**], [**?**].
7) *Distributed Consensus and Synchronization:* Hashicorp Serf, Apache ZooKeeper, Google Chubby, Doozerd, etc.
8) *Host Remote Access:* Fabric SSH

IV. Qualitative Evaluatiion of InitSDN

In this section we provide a qualitative evaluation of InitSDN's capabilities. A rigorous quantitative evaluation is part of our future work. In evaluating InitSDN qualitatively we focus on properties such as the ease of performing some of general use cases for the management of SDN control plane with and without InitSDN.

A. Evaluation Criteria: Building Network Applications for SDN Control Plane Management

This criteria is relevant to the SDN service providers. As we discussed in the previous section, InitSDN separates the control and meta-control messages. This helps to modularize the network applications by providing separation of concerns between two different types of applications as follows:

1) *SDN network application:* These are the network applications that instrument the network among the hosts. These are developed by the SDN user or vSDN(virtual

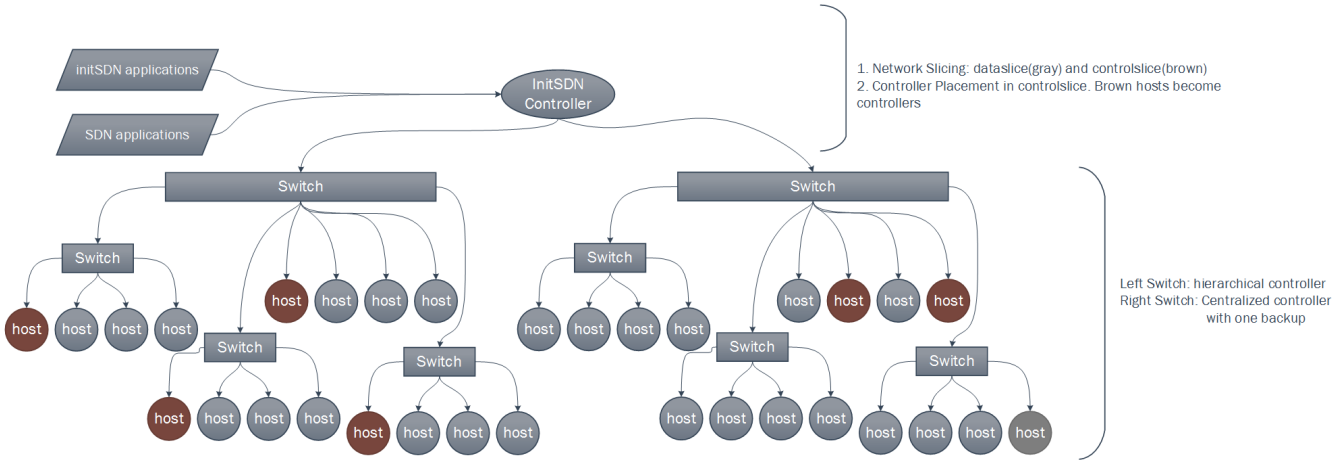

Fig. 2. **Legacy Network During the Bootstrapping Phase**
(1) network slicing step has been executed (2) Brown colored hosts are chosen to be in the control plane as per topology and configuration

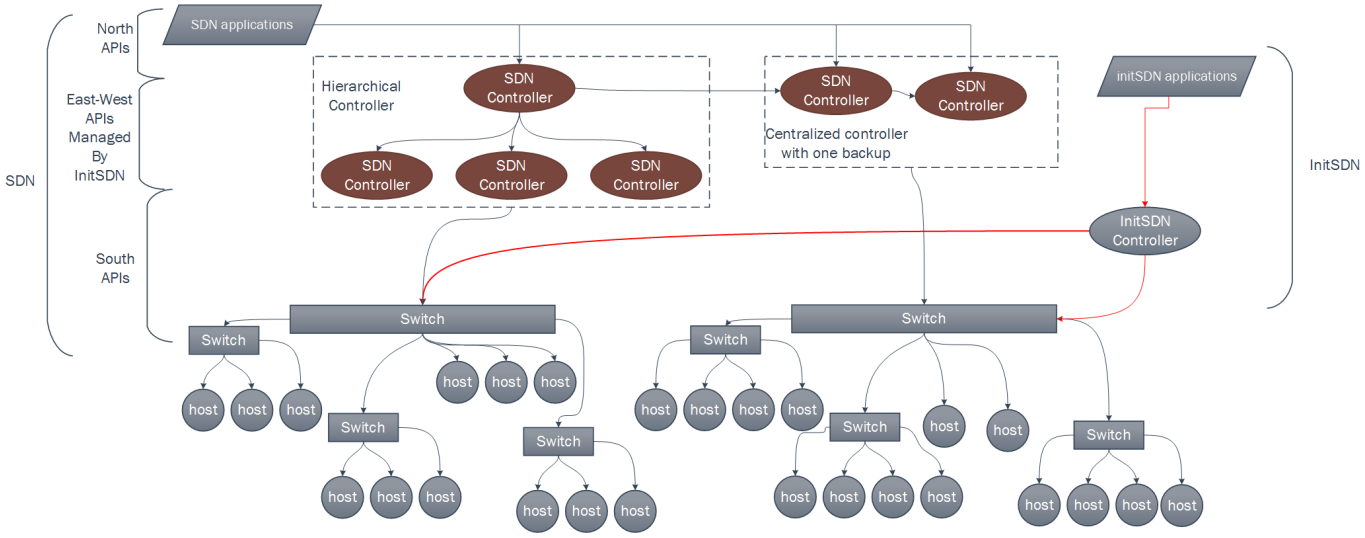

Fig. 3. **Legacy Network turned into SDN Network After Bootstrapping is Completed**
(1) InitSDN has taken a back seat (2) SDN controllers are placed in control plane, configured and have been activated

SDN) tenant. Examples of such applications are routing (OSPF, IS-IS, BGP etc), security, access control, application-based forwarding, etc. These applications are written against the controller that client is using in its SDN (or vSDN).

2) *InitSDN network application:* There is another type of application that instruments the network along the control-plane. These are developed by the SDN service providers. Examples of such applications are switch migration, controller migration, VM network state migration, control-plane scale up/down, controller updates, control-plane topology management, vSDN control-plane management, etc. Without InitSDN, these applications have to be written for individual controllers. For example, if SDN hosts three types of controllers, then the controller migration application has to be written for each of these controllers. However, these applications become easier to develop with InitSDN since such applications now need to be written against only InitSDN irrespective of the number of controllers, number of vSDNs, or types of controllers present in the system.

In this way, InitSDN brings the separation of concerns in the SDN control plane management.

B. Evaluation Criteria: Controller Scale-up/Scale-down

Controller scale-up or scale-down can be achieved easily using InitSDN.

1) *scale-up:* InitSDN needs to find out idle hosts (or VMs) for adding them to the control-plane. This has to be programmed by a network operator through the InitSDN application. InitSDN then adds such new hosts to the control plane. InitSDN installs controllers on these new hosts. It also modifies flow-rules on new switches, so that they start to redirect their traffic to new controllers.

2) *scale-down:* InitSDN simply modifies the flow rules in the switches to point them to controllers from to be scaled-down control-plane only. After that InitSDN can either shutdown hosts containing extra controllers (i.e. those controllers which are now not connected to any switches) or use them for other controllers (e.g. different vSDN).

This way InitSDN provides scalability to the SDN control plane. This also increases reliability of SDN control plane against network load changes.

1) Make InitSDN build a new topology.
2) Compare old and new topology and find out the scale up/down steps required.
3) Ask InitSDN to scale up/down accordingly.

C. Evaluation Criteria: Controller/switch Migration

In InitSDN, the controller or switch migration is reduced simply to the task of updating the control-plane topology. InitSDN builds new control-plane topology after notified by its discovery module about the change in the network topology. This new topology is then enforced on the control plane as described in the previous subsection.

V. Related Work

The authors in [**?**] propose a solution called the Pratyaastha control plane to address a related but different controller placement problem. Pratyaastha first partitions the SDN application state into the lowest granularity possible so that it can be distributed across the controllers. Subsequently, based on the controller load, it decides the placement (in this case, reassignment) policy that maps the switches and application state to the out-of-band controller instances. This placement problem is different than ours where the controllers are mapped to the physical hosts. Hence, Pratyaastha does not require the network hypervisors since the control plane still resides on the dedicated network. Though, Prayaastha provides elasticity to the control plane in the case of changing controller load, it does not provide elasticity in the case of major network topology changes or large-scale failures in the initially assigned control plane physical hosts since the control plane physical nodes are still statically assigned.

The authors in [**?**] describe a two-level controller hierarchy called "Kandoo" with the lower-level controllers being responsible for handling the frequent events and short-lived flows, while top-level controllers handle the other flows. However, it is not flexible enough to adapt to the network topology and load, e.g. in the case where most (or all) of the network flows are long-lived. The authors in [**?**] discuss the placement problem in the control plane and observe that a single controller is sufficient for most of the use cases. However, it does not consider the use case that requires robust fault tolerance, virtual SDNs, multi-level controller hierarchy, etc where multiple controllers are needed and hence placement becomes more complex. Another effort [**?**] discusses the controller placement problem but in the context of the network load alone. It does not provide configurable control-plane topology.

VI. Conclusion & Future Work:

In this paper we highlighted the limitations of the current SDN distributed control plane in terms of controller complexity, reduced flexibility, scalability and reliability. To address these concerns, we described a solution approach that involves a separate bootstrapping or initialization phase for the SDN network. Our solution is called InitSDN and its architecture involves a number of functionalities that relate to topology, discovery, synchronization, and placement. Our current work has qualitatively evaluated the benefits stemming from the work in terms of ease of developing the controller logic and operationalizing the SDN network for network operators using real world network toplogies.