SPARC: Towards a Scalable Distributed Control Plane Architecture for Protocol-Oblivious SDN Networks

Mingzheng Li, Xiaodong Wang, Haojie Tong, Tong Liu, and Ye Tian*
School of Computer Science and Technology, University of Science and Technology of China
E-mail: {salmz,qsmywxd,twdlll,atong}@mail.ustc.edu.cn, yetian@ustc.edu.cn

***Abstract*—High-level programming abstraction and large-scale deployment have become two important trends of software-defined networking (SDN) in the past decade. Using high-level program to manage the large-scale network faces more serious control plane extensibility problem due to its complex intermediate representation and protocol-independent feature. To address this problem, we propose SPARC, a programmable and scalable controller architecture, which employs a hybrid hierarchical structure to maximize flexibility with regard to control plane distribution. Our architecture also allows for pushing down control decision making closer to the data plane and localize network event processing to lower the latency of control plane operations while exploiting SDN's global visibility to build optimal policy decisions. Furthermore, we investigate the feasibility of SPARC by exemplifying the case of delivering ICN mobility services and then conduct evaluations to demonstrate the efficacy of our design.**

***Keywords*—SDN, high-level programming, large-scale, pushing down control decision.**

I. Introduction

As an new network paradigm, Software-Defined Networking (SDN) helped decouple the control plane that describes network functionalities with high-level languages, and the data plane that forwards packets based on low-level rules. Over the fast few years, many programming models have been proposed to raise the abstraction level of network devices [1], [2], [3], [4], [5]. On the other hand, emerging programmable hardwares (e.g., RMT and Flexpipe) allow both fine-grained control and high-speed packet processing, and are being increasingly deployed in different platforms [6]. A compiler is required to bridge the given program and target hardwares. Typically, the compiler first transforms the network policies into intermediate representations, and then maps the latter into forwarding rules installed in SDN switches from top-down [7], [8], [9].

However, with the addition of reconfigurable switches, the intermediate representations maintained in the control plane increase dramatically, and accordingly the controller may suffer from scalability issues and become overloaded. In fact, the scalability problem of SDN networks attracts considerable attention from academic researchers, and various controller structures have been put forward to upgrade the control plane processing capacity [10], [11], [12], [13], [14]. Unfortunately, these existing solutions can not be directly portable to the controller with adequate abstractions. We argue that the following two challenges should be considered from the perspective of managing a large-scale programmable network via high-level control programs:

1) High-level network policies usually treat the entire network as a whole and ignore substrate details. Such abstract representations need to be translated into specific flow rules installed on distinct switches [15]. This compiling process is expensive and time consuming, especially when programmers adopt event-driven style to handle packet misses. Hence, high-level programming may face more serious scalability problem compared with traditional multi-controller environments.
2) The reconfigurable switches should be programmed to recognize new protocols and to support customized control logic. But, former extensible controller technologies are mainly designed for OpenFlow, verify only specific packet header fields, and can not interoperate with the programmable data plane [13], [14]. PNPL [16] and P5 [17], which are implemented based on POF and P4 respectively, provide two frameworks for managing protocol-independent data plane, but neither of them are suitable for dealing with scalability issues.

In order to address the above two challenges, we propose SPARC, a **S**calable and **P**rogrammable control plane **ARC**hitecture for protocol-oblivious SDN network. Inspired by previous extensible control plane attempts, SPARC adopts two-layers deployment of SDN controllers that handle frequent, localized events close to data plane by pushing down decision making, in the form of an intermediate representation named *extended tracing tree (xTT)*, to the regional controllers. Global events, which require a broad picture of network-wide state, are handled by upper layer global controllers. Following the idea of PNPL [16], SPARC provides a set of *programmatic APIs* that allows a programmer to describe algorithmic network policies upon self-defined network protocols. This high level user program is data plane agnostic and will be compiled into separated xTTs in the global controller layer. On the other side, the near-sighted regional controller maintains a *runtime system* that will timely produce forwarding rules by searching the constructed partial tree in memory. To sum up, the main contributions of this article are as following:

- We have designed and implemented an initial prototype

*Correponding author.

This work was supported in part by the National Natural Science Foundation of China (No. 61672486) and the subtask of Strategic Priority Research Program of the Chinese Academy of Sciences (No. XDC02010703).

of SPARC that provides protocol oblivious programming environment and effectively solves the control plane scalability problem at the same time.
- Unlike previous works attempt to distribute the network policy to several controllers, we present a novel technology for timely reacting local network events by pushing down decision making (i.e., the compiled intermediate representation) closer to the data plane. Our approach can significantly reduce the control plane response time and balance the total workload.
- We validate the feasibility of SPARC by deploying Information-Centric Networking (ICN) with inherent mobility support, and demonstrate how mobility-as-a-service (MaaS) can be realized over a converged ICN/SDN network. Moreover, we carry out performance and scalability evaluation of ICN use case by scaling up/down the number of managed switches.

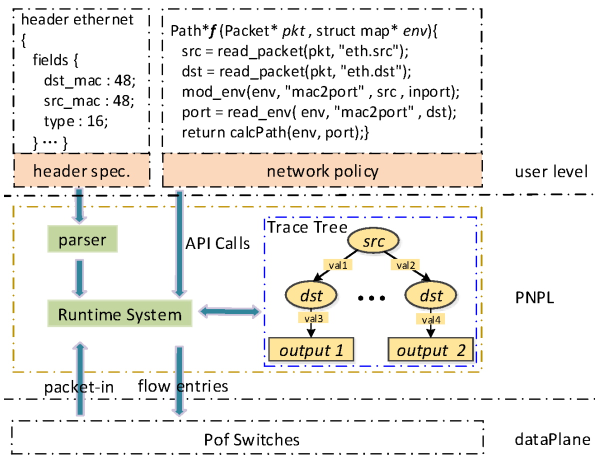
The remainder of this paper is organized as follows. We first offer a brief introduction to PNPL and summarize related works in Section II. Then, we detail the design of proposed architecture in Section III. In Section IV, we showcase the use of SPARC for delivering ICN services and discuss the evaluation results. Finally, we draw conclusions in Section V.

II. Background and Related Work

A. PNPL Primer

SPARC is built upon our previous PNPL framework [16], which provides an easy-to-use paradigm for controlling the protocol-independent data plane. Specifically, PNPL provides a header specification language and a set of protocol-agnostic programming APIs. The header specification allows a user to define complicated packet header structures through a high-level abstraction language. PNPL's APIs offer an implicit call of POF instructions, simplify programming since users do not have to appoint instruction parameters. With PNPL, a programmer can define arbitrary network protocols, and describe algorithmic sequential program in an ***f*** function.

On receiving a *Packet-In* message, PNPL invokes user programs, monitors its execution process and further generates an execution trace that records both packet parsing steps and API call sequences made by ***f*** function. PNPL's runtime system then constructs a data structure named *extended tracing tree (xTT)* by incrementally merging historical execution traces into a rooted tree. The runtime system is also responsible for producing forwarding pipelines from the constructed xTT and deploying them on POF switches. More details about xTT construction and pipeline generation can be found in [16]. We use basic *switch_learning* function to illustrate key features of PNPL programming model. As shown in Fig. 1, PNPL takes an *Ethernet* protocol specification and an ***f*** function (including two arguments, new arriving packet and the network environment) as inputs. For learning switches in Fig. 1, ***f*** maintains a `mac2port` table that maps a switch, MAC address pair to corresponding output port. Specifically, ***f*** first reads necessary packet fields (`inport`, `srcMAC`,



```
header ethernet
{
  fields {
    dst_mac : 48;
    src_mac : 48;
    type : 16;
  } ··· }
```

```
Path*f (Packet* pkt , struct map* env){
  src = read_packet(pkt, "eth.src");
  dst = read_packet(pkt, "eth.dst");
  mod_env(env, "mac2port" , src , inport);
  port = read_env( env, "mac2port" , dst);
  return calcPath(env, port);}
```

Fig. 1: PNPL Architecture.

`dstMAC`), and then records the mapping relationship between the `srcMAC` and `inport`. Next, it lookups the `dstMAC` in `mac2port`, returns a forwarding path by using CalcPath() function. The xTT formed after applying ***f*** to given packets is clearly containing all policy execution flows and relevant policy decisions made by user function. Accordingly, the generated pipeline matches `srcMAC` in one table for learning, and matches `dstMAC` in another table for forwarding.

B. Related Work

1) Deep programmable SDN data plane: Recent progress on programmable switch allows custom protocol stacks as well as flexible packet processing by means of reconfigurable *match+action* pipeline. To describe this pipeline, P4 [18] provides an abstract model with five parts including headers, parser, actions, tables and control flow. POF [19] strengthens the flexibility of pipeline structure by supplying a set of general-purpose instructions that allow a programmer to dynamically adjust deployed network protocols and applications on the fly. OpenDataPlane [20] is a bottom-up open-source project that pursues a set of unified APIs covering common features across various networking platforms.

2) High-level SDN programming languages: Multiple systems have been proposed for offering programming languages with different levels of abstractions. The Frenetic family (e.g., Frenetic [1], Pyretic [2]) is one of the most influential studies in this field and has derived many effective SDN programming languages. Frenetic and its extension provide an SQL-like query language for declaratively expressing network policy. Pyretic introduces modular compositors and permits parallel and sequential composition of user policies. NetCore [3] is designed to replace Frenetic with new formal semantics and correctness proofs. Concurrent NetCore [4] adds multi-table support but still depends on manual definition of flow table layout. Stateful NetKAT [5] and SNAP [21] leverage persistent switch-local state to enable stateful packet processing. Apparently, the control plane scalability issue is not within the scope of these study.

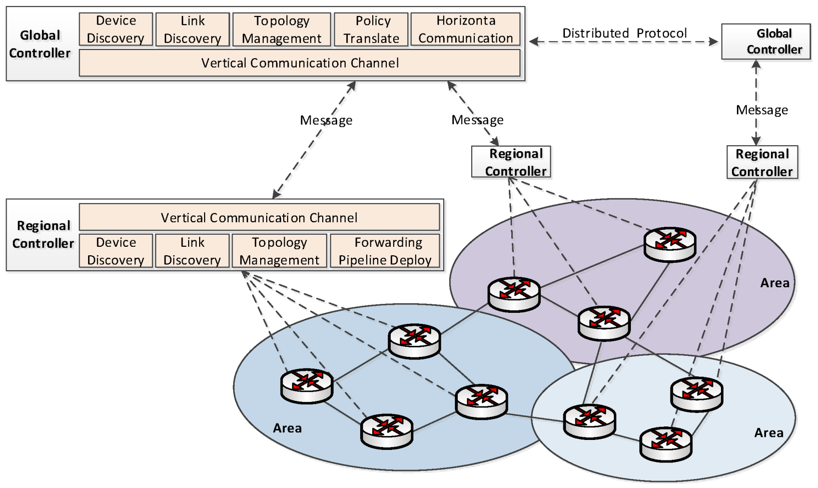


Fig. 2: SPARC's Hybrid Hierarchical Architecture.

3) Scalable SDN control plane: Existing approaches to alleviate the control plane scalability problem can be divided into two categories. The first category approach distributes the administration task to multiple controllers. HyperFlow [10] is the first distributed event-driven controller architecture that supports OpenFlow. It synchronizes global network view among controllers from different clusters through a publish/subscribe system. DISCO [11] is another extensible SDN control plane with the capability to differentiate heterogeneous links of modern overlay networks. ONOS [12] acts as a special network OS supporting horizontal scaling, and performs better in network availability and reliability. The other kind of approach uses hierarchy structural design. For example, Kandoo [13] processes local applications on local controllers near datapath, leaving global applications to remote root controller. Orion [14] adopts a hybrid hierarchical architecture, and is the most similar literature with our proposed SPARC.

Note that all these works are based on OpenFlow with limited protocol compatibility, and may be not directly applicable to SPARC. Nevertheless, they provide useful tips for enhancing the control plane extensibility, and more generally building a logically centralized but physically distributed SDN network.

III. The SPARC Design

In this section, we present the design of SPARC, a scalable and programmable control plane architecture of large-scale SDN networks. We first give a high-level overview of the proposed network architecture as well as its building blocks, and then describe the key structural and functional components of the controllers in more details.

The overall structure of SPARC is demonstrated in Fig. 2. From the figure, we can see that SPARC divides the entire network into disparate network domains, where each domain in turn consists of multiple areas with general-purpose forwarding devices. Moreover, SPARC adopts a two-tier controller deployment with two types of controllers: the Global Controller `GC` at the upper layer and the Regional Controller `RC` located close to the data plane. In the following, we describe their functionalities respectively.

Global Controller (`GC`): Each `GC` is in charge of an SDN domain, maintains its own network state and constructs a global network view using synchronization information from other `GC`s. In addition to this east-west communication interface, a global controller also inherits the original northbound interface of PNPL, i.e., a header specification language and a set of programmatic APIs, with which users can freely compose network policies over self-defined protocols. The `GC` translates the user program into an intermediate representation *extended tracing tree (xTT)*, and further separates it into subtrees based on areas and sends them to corresponding regional controllers. To minimize control overhead, we enable the `GC` to exclusively handle events which require network-wide visibility or have not been traced by `RC`'s runtime system. Examples include inter-domain routing, mobility tracking or rule installation for newly arriving flows.

Regional Controller (`RC`): The regional controller, which operates inside an area, is responsible for gathering network resources exposed by physical devices and links, and further reports collected local information to the global controller. Hence, there is no need to exchange information among `RCs` directly, which helps decrease the route convergence time when POF network scales to large size. Moreover, each `RC` has a runtime system that is used to cache the `GC`-generated *partial tree* (henceforth referred to as *PT*) in memory. When a *Packet-In* event occurs, the runtime system first handles the reported packet by searching it in the PT. If the search process reaches a leaf node with policy decisions, the runtime system can immediately produce and deploy forwarding rules that enforce the deserved network policy. Otherwise, `RC` will send the packet (without payload) to remote `GC`, and incrementally update local PT with new branch returned by `GC` for handling the subsequent packets.

This two-tier approach allows us to improve the controllability of POF network. In SPARC, the packet handling requirements are tried to be satisfied by regional controller with the aid of PT. The RC's runtime system caches all the previous policy decisions from a fine-grained local scope and lowers the latency of control plane operations by directly retrieving memory contents. Therefore, it can handle frequent, load-intensive *Packet-In* events when flow entry timeout or replacement happens at the data plane. Put differently, we offload part of critical tasks from the top-tier controller to the RCs, and this can significantly reduce the load on the GC and further enhance the scalability of control plane.

A. Basic Components

As shown in Fig. 2, SPARC contains several basic components, which are essential for realizing the intra-domain information processing and inter-domain information processing. We further explain them in more details.

1) Device Discovery Module: The Device Discovery Module stores sufficient information regarding network devices, i.e., switches and hosts, in the data plane. The hosts are normally user terminals or virtual machines connected to the POF switches, while the latter are protocol-agnostic packet processors with generic rules. Note that this module includes two sub-modules: (a) The Regional Device Discovery Sub-Module discovers the existence of POF switches in its area during the initial handshaking process, obtains the intra-area host information (e.g., network address or entity name) through the location announcement packet, such as the *ARP* packet in IP network and the *Publish* packet in ICN network. (b) The Global Device Discovery Sub-Module records the abstracted information of all switches, such as the datapath identification (DPID) and the number of ports. By collecting the host information from each region, it also builds a profile for other modules to facilitate inquiries.

2) Link Discovery Module: The Link Discovery Module leverages the Link Layer Discovery Protocol (LLDP) to detect the inter-connected link between the POF switches. Specifically, the RC encapsulates an LLDP packet with its unique fingerprint in a *Packet-Out* message, and sends it to each port of all local switches. Upon receiving this LLDP packet, a neighbor switch forwards it to the controller in the form of *Packet-In* message. Then the RC decapsulates the *Packet-In* message and updates its link set based on the meta information (e.g., DPID and ingress port ID). If the extracted fingerprint differs from its own fingerprint, the RC knows that this link is an inter-area link and the reported switch is an edge switch, then it sends the original *Packet-In* message to GC. The latter recognizes all area border switches, and stores the inter-area/-domain link information including the interface connected to neighbor areas, maximum bandwidth, etc.

3) The Topology Management Module: The Topology Management Module has two parts to create a complete topology view of the entire network. (a) The Regional Topology Management Sub-Module requires discovery of the network topology of the assigned area by collecting infrastructure information from the previous two modules. Meanwhile, it also adopts a resource abstraction scheme to shield the bottom details when reporting its area network view to GC. In particular, the reported abstract topology consists of:

- DPIDs of all edge switches and host-attached switches.
- Path capacity information about two kinds of virtual link, such as hops and bandwidth, etc.

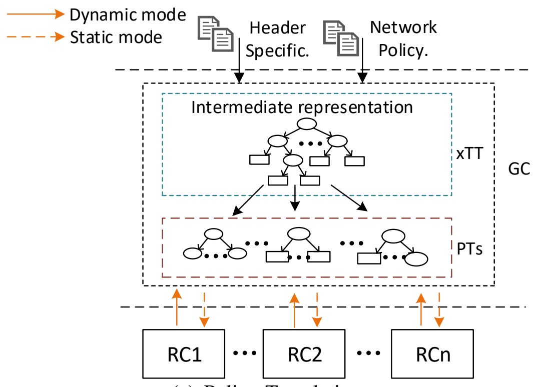
Here, the first kind of virtual link means the pre-computed path (with shortest hops) from one inner switch to one edge switch, while the second abstract link is computed between any pair of area border switches. (b) The Global Topology Management Sub-Module enriches the area-level network view by adding the neighbor area relationships drawn from the received LLDP packets. It further obtains all inter-domain knowledge on network topology through the below-mentioned Message Communication Module, and finally reaches a synchronized network-wide view.

4) Message Communication Module: The Message Communication Module has two sub-modules which are responsible for the horizontal network information synchronization and the vertical xTT distribution respectively. On one hand, we use the the Advanced Message Queuing Protocol (AMQP) [22] to provide a publish/subscribe communication channel between neighboring GCs. With AMQP, each GC shares its abstract domain information (including all hosts, switches and topology information) via network-wide flooding, supports the cross-domain communication by sending and receiving segment routing requests. It also uses Keep-Alive messages to test the presence of peering GCs. On the other hand, we create a vertical communication channel between the GC and the RC with asynchronous socket. The asynchronous socket has a thread pool that calls relative modules to handle the incoming network events. Examples include: topology updates when link failure occurs and xTT distribution upon receiving new arrival flows.

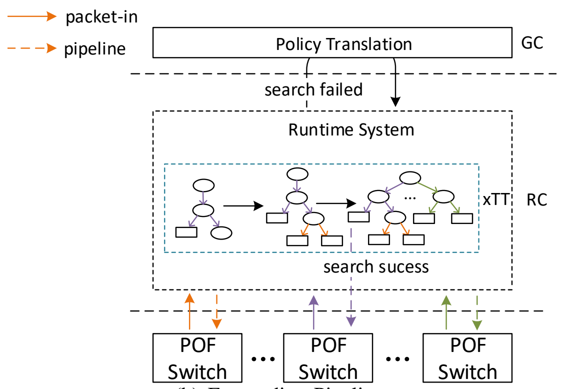
B. Core Compiler

The Core Compiler is the heart of SPARC and encompasses the *Policy Translation* and the *Forwarding Pipeline Deployment* modules. The network policies received from user programs are compiled and split into distinct trace trees for automatically populating the multiple flow tables on the target area switches.

1) Policy Translation Module: In SPARC, a programmer can declare his/her network policies through the appropriate notations and abstractions inherited from PNPL. However, such a high-level construct can not be understood directly by other controller modules. Hence, as shown in Fig. 3(a), the Policy Translation Module operates as a transpiler that takes user programs as input and produces the equivalent intermediate structure xTT. As a result, the translated xTT encodes the same packet parsing and network policy functionality as the source code of user programs. The constructed xTT will be further partitioned into PTs, which are stored in localized RCs dispersedly. It is worth mentioning that this translation process can be triggered in the following three modes:



(a) Policy Translation.



(b) Forwarding Pipeline.

Fig. 3: Core Compile Components.

- Dynamical mode–In the dynamical mode, when a new packet arrives, the GC invokes user program to augment its xTT with a new growth branch, and distributes the additional branch to appropriate RCs. Here the new packet might come from *Packet-In* messages or routing requests sent by other GCs.
- Statical mode–In the statical mode, the GC knows all traffic flow beforehand by keeping a traffic history, and use it to build an extensive xTT for installing forwarding rules in advance before network flows arrive at the switches.
- Hybrid mode–In the hybrid mode, the GC combines the advantages of both reactive and proactive approaches, pre-produces an xTT for certain traffic flows and then modifies (add/update) it in case of network events like link failures and user mobility.

Another noticeable thing about our transpiler is the compilation of *return* statement declared in the code of network policies. From programmers' point of view, the execution of user-defined `CalcPath()` function outputs a forwarding path and returns a valid action, e.g., drop or forward packets to specific port, for each switch along the path. What really happens in SPARC is that the GC takes a centralized view of the network state and makes hierarchical forwarding decisions to determine optimal traffic routing. Specifically, a GC first reads the source address and destination address from the new arriving packet. (a) If both source and destination address are in the same region, GC then assign the route computation task to corresponding RC by sending a request message $\langle$IngressDPID, IngressPort, EgressDPID, EgressPort$\rangle$. (b) If the two address are from the same domain but different regions, GC uses the Dijkstra algorithm to compute a shortest path based on the sum of stored link result, then breaks up the routing path into segments and sends request messages to RCs in a reverse sequence manner, i.e., destination, transit and source areas. (c) If the two communication hosts locate at different domains, GC calculates a global shortest path by adding inter-domain hops and intra-domain hops together, and then uses the horizontal communication messenger to send the original packet and route request to every domain along the path for triggering the translation process on other GCs.

2) Forwarding Pipeline Deployment Module: As shown in Fig. 3(b), the Forwarding Pipeline Deployment Module is responsible for producing dynamic and efficient multi-table pipelines and installing them on POF switches. The central component is the runtime system in which each RC stores all the previous policy decisions using the PT maintained in working memory. Since the space in flow tables is finite, switches can only store limited flow rules. Frequent *Packet-In* events will occur in the heavy traffic situation due to the existence of per-flow timeouts and rule replacement mechanisms in SDN. We argue that the processing of such events should be placed as close as possible to where they originate by relying on regional controllers. To handle a new reported packet, RC's runtime system simply traverses the PT from root node. (a) If the policy decision for this packet has already been traced, then RC can automatically infer forwarding pipelines from the PT. (b) Otherwise, RC sends the packet without payload to the connecting GC to invoke user programs, and augments its PT with the returned branch for subsequent packet handling.

We keep the network control logic centralized and localize control decision making to each RC to minimize control plane response time. Unlike GC that explicitly invokes policy programs to handle the *Packet-In* packet, RC directly search the PT in memory, which is relatively faster in terms of execution time. Moreover, latency-critical or high-load tasks are offloaded from the global control plane to the RCs. This can significantly decrease the load on the GC, such as computational complexity of inter-domain routing. It can also reduce the latency because there is no need to consult the GC on each arriving flow.

IV. Enabling ICN Services with SPARC

As an innovative network architecture, Information-Centric Networking (ICN) shows great potential to solve issues in traditional IP network, such as mobility, security, energy efficiency, etc. Meanwhile, the flexible programmability and manageability of SDN bring convenience to ICN deployment

[23]. However, a simple combined ICN/SDN solution faces more serious control plane scalability problem because content resources are highly dynamic. In this section, we reveal the additional benefit of SPARC from the viewpoints of integrating ICN into POF network, and conduct simulation experiments to demonstrate its performance advantages.

A. Name-Oriented Routing

Our preliminary implementation of the ICN controller in SPARC is fully compliant with PNPL's language specification. For the sake of brevity, we employ a simple design of ICN packet format. For example, as shown in Fig. 4(a), the outmost *Ethernet* packet header is retained to be compatible with existing hardware devices. The *ICN_Protocol* field (4 bits) is used to distinguish ICN protocol from multiple other protocols like IP. There are four basic types of ICN packets: Advertising, Interest, Data and Heart-Beat. Among them, Interest packet and Data packet have an extra variable-length field for storing a global *Entity Unique Identifier (EUID)*, which represents a piece of registered content. Here, each EUID can be either a flat self-certifying name or a hierarchical human-readable name. The remaining *Payload* part of ICN packet includes signatures, user keys and data, etc. In Fig. 4(b), we also

```
header ethernet {
  fields {
    ...
    type}
  next select (type){
    case 0x0800: ipv4;
    case 0x0901: icn;}}
header icn {
  fields {
    euid : 48;
    type : 4;} ··· }
```

(a) Self-defined ICN protocol.

```
Path*f (Packet* pkt , struct map* env){
  port = read_env( env, "port");
  if(search_header(pkt, "eth.type", 0x0901)){
    type = read_packet(pkt, "icn.type");
    euid = read_packet(pkt, "Icn.euid");
    if(type == associationEvent){
      mod_env(env, "location", port, euid);}
    if(type == mobilityEvent){
      mod_env(env, "location", port, euid);
      return calcPath();}
    if(type == contentRequestEvent)
      return calcPath();}}
```

(b) ICN routing and mobility service.

Fig. 4: ICN Service.

schematize the high-level control plane program that enables network routing in ICN-compatible POF switches [24]. According to program order, the whole content-driven routing process can be divided into two stages. (a) End hosts first actively report local content resources to their RC by sending advertisement packets with EUIDs. The RC further notifies its GC if new EUIDs are registered. Note that when an EUID is no longer available in local area, the RC has to evacuate previous report. (b) In the second stage, a user issues a request with destination EUID to obtain his/her interested data. If the first-hop access POF switch has already cached the corresponding content, the user can retrieve a data copy directly. Otherwise, the switch will forward Interest request to its RC, and the latter will ask a GC for help. Then, the GC chooses the best content provider and plans a reverse path to transfer data packet to the requester. It also makes caching decisions on intermediate network nodes along the path.

B. Mobility as a Service

ICN uses location-irrelevant name to recognize a network entity, it brings built-in mobility support since end-user's entity maintains a persistent name when moving around. In the following, we consider the case of delivering mobility-as-a-service (MaaS) over the SPARC-programmed network, which has an additional benefit of being exceptionally easy to implement (see Fig. 4(b)). Specifically, we give two examples to illustrate how SPARC carries out the inter-area mobility management and inter-domain mobility management.

1) Inter-area Mobility Example: The inter-area mobility management is performed as shown in Fig. 5. The adopted topology contains one domain and the domain has three areas. Both content requester and provider are in the same region (R1) at the beginning. When the provider moves from R1 to R2 within the same domain, the RC which connects to R1 generates a *Packet-In* message to de-register the binding relationship between EUID and the attachment switch. Meanwhile, R2's regional controller reports the provider's current location to the GC it connects, the latter then forges an Interest packet and recursively calls the user program to re-calculate a path for delivering content on the updated content information. Since the newly deployed flow rules have higher priority, the original rules will eventually expire after a timeout period. By comparison, the requester move case would be much more easily achieved due to ICN's late-binding [25] feature which allows controller to redirect flows to requester's new position in a dynamic manner.

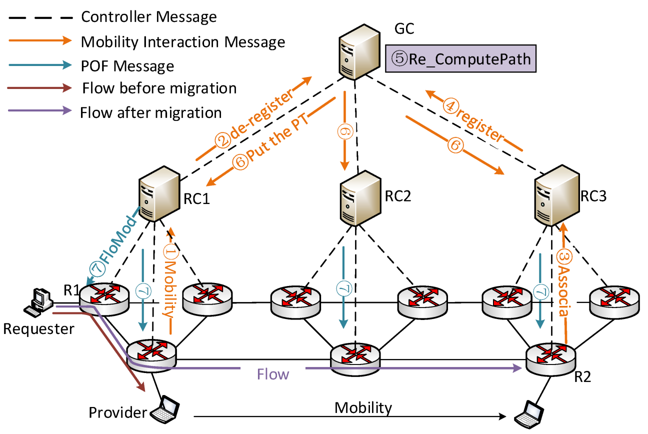


Fig. 5: Inter-area mobility scenario.

2) Inter-domain Mobility Example: As shown in Fig. 6. The adopted topology contains two domains and each domain has two areas. The second example shows how to maintain a persistent communication session between the requester and provider even when they move from one domain (D1) to another domain (D2). Despite the control program here is the same as the one in inter-area mobility case, the actual operation applied during the movement is different. Specifically, when regional controller1 announces the departure of content provider from R1, D1's global controller encapsulates the relevant EUID to a simple request and publishes it to all

GCs. When global controller2 which has content provider's current location receives the request, it computes an area-level path and sends messages to GCs on the routing path, the latter further pass messages to specific RCs. Here, let's assume path R1-R2-R3-R4 is chosen. When corresponding regional controllers receive the message, they extract ⟨IngressDPID,

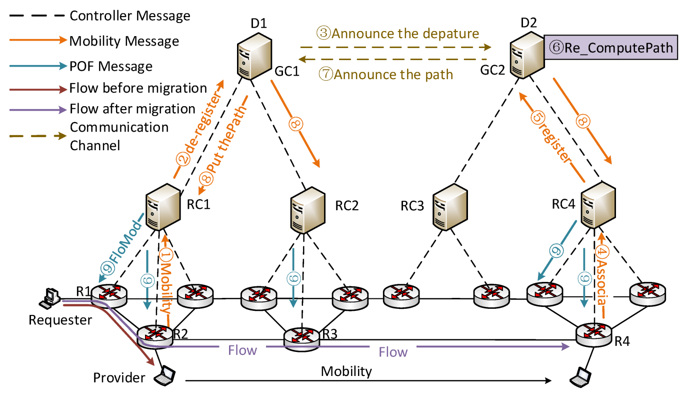


Fig. 6: Inter-domain mobility scenario.

IngressPort, EgressDPID, EgressPort⟩ and distribute forwarding rules on switches along the intra-area path. In addition, the content requester mobility can also be conducted in a similar way.

V. Implementation and Evaluation

In this part, we present the implementation of SPARC and realize the described ICN services on top of this framework. Then, we evaluate the overall performance of our prototype system in various simulation experiments.

A. Implementation Details

We have developed a SPARC prototype to validate the feasibility and effectiveness of the proposed system design. The implementation of SPARC is based on PNPL, which provides basic topology management, storage and POF message modules. We use hierarchical design idea to split the policy compilation functionality of PNPL into two parts, which are correlated to the policy translation module in upper layer GC and the pipeline generation module in bottom layer RC. The communication between this two modules is through SPARC's vertical communication channel. To sustain network scalability, we adopt a distributed domain-based approach to increase the coordination between GCs. The constructed regional controller and the global controller of SPARC has more than 25,000 C/C++ source lines of code (SLOC) respectively. Moreover, we have also implemented an ICN routing and mobility manager (270 SLOC) as an add-on module of SPARC. This application gathers both network resources and content resources, queries the topology module to compute a path for content transfer on receiving interest requests.

B. Testbed Setup

We build a real-world testbed with four interconnected Linux servers, where each server has two Intel I7-8700 cores

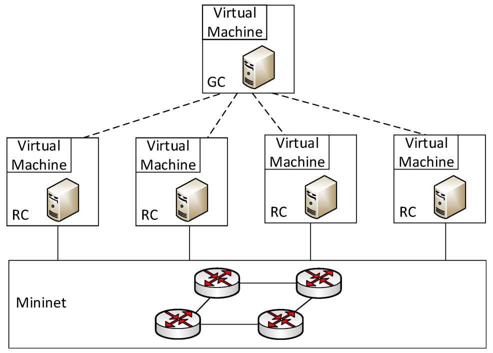


Fig. 7: The single domain experimental testbed.

(3.2 GHz CPU, 16G RAM) and 256GB memory. Fig. 7 demonstrates the experiment environment considered in a single server. We utilizes the VMware virtualization technology to provide several virtual machines, each server runs one global controller and four regional controllers. Meanwhile, on each server, we use Mininet [26] to instantiate a random network topology with N POF switches, N hosts (connecting to each switch) and M links. We consider a set of N^x contents uniformly distributed throughout the simulated network (x is a variable). The access frequency of content follows a Zipf distribution, that is, the i^{th} popular content will be requested with probability of $f(i) = c/i^{\alpha}$. The arrival process of interest request at control plane is a Poisson arrival process. In the simulation, we choose $N = 250$, $M = 500$, $x = 1.8$ and let the content priority between 0 and 1000. The constant c and exponent parameter α in $f(i)$ is set as 1 and 0.5 respectively.

C. Performance Evaluation

1) Comparison of Response Time: Firstly, we evaluate the effectiveness of the proposed pushing down decision making strategy, and compare with other controllers including PNPL [27], and Floodlight [28]. Specifically, we insert code in POF/OpenFlow switches and measure the control plane response time, i.e., the latency between sending table miss *Packet-In* requests and receiving forwarding rules produced by controllers. Since Floodlight can not process ICN packets, we use *switch_learning* as the control program. One Linux server with 250 switches is involved in this test, and we employ Pktgen [29] to construct 5000 packets from switches. Additionally, we start one GC and four RCs in SPARC case.

We repeat this experiment for five times and present the results in Table. I, where the lowest value in each row has been highlighted. From the table we can see that SPARC has a larger flow setup latency in the first trial and performs much better than the other controllers in the following test. The observation is easy to understand. For SPARC, all packets are sent to global controller for making policy decisions once generated for the first time, which will bring extra delay due to the vertical communication overhead. After the 1^{st} test, previous policy decisions will be traced in the form of PT at regional controllers. Therefore, SPARC can directly handle subsequent packets with the same policy decision by searching them in

TABLE I: Control plane response time comparison.

Controller	SPARC(us)	PNPL(us)	FloodLight(us)
Exp1	5.61x10^8	**4.25x10^7**	4.92x10^{11}
Exp2	**2.57x10^5**	2.53x10^6	4.83x10^{11}
Exp3	**2.61x10^5**	2.45x10^6	4.85x10^{11}
Exp4	**2.57x10^5**	2.55x10^6	4.96x10^{11}
Exp5	**2.59x10^5**	2.66x10^6	4.87x10^{11}

PT without invoking user programs. Apparently, searching PT in memory is executed faster than explicitly calling program as done in Floodlight. The better performance of SPARC over PNPL can be explained with the fact that PNPL has to deal with 250 sockets from switches simultaneously and SPARC exploit the parallelism in multi-controller environment.

Since SPARC benefits from PT caching, people may have a concern about whether it will impose large memory overhead. To answer this question, we plot the consumed memory of all SPARC controllers in Fig. 8, from which one can see that the memory increases within an acceptable limit (80MB).

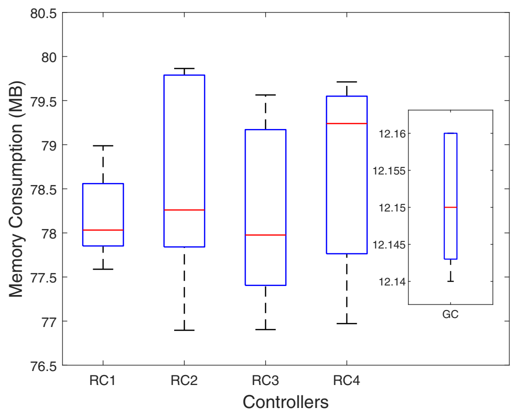


Fig. 8: Memory consumption of SPARC with five experiments.

2) Scalability Evaluation: In the second experiment, all Linux servers with 1000 switches in total are used. We increase both the number of regional controllers in each domain as well as the number of domains to verify the scalability of SPARC. We run the ICN routing and mobility module on top of SPARC, and use Pktgen to simulate content announcement and interest requests from data plane. Note that the total amount of content resources N^x is approximately 20,715 under the given parameter configuration, and is far greater than the amount of network devices, which reflects that SPARC will face more severe scalability problems.

We evaluate the impact of area and domain partitions on request handling rate (i.e., flow setup rate) per second of SPARC. Fig. 9 presents the experimental result, where the number of employed GCs and RCs is denoted in the form of g-r in the abscissa. From this figure, we can make the following two observations. First, when there is only one domain, a single global controller has to handle all requests with a number of 39735 per second. Along with the increase of divided areas, the maximum throughput of SPARC grows at a near-linear trend. That is because more tasks can be

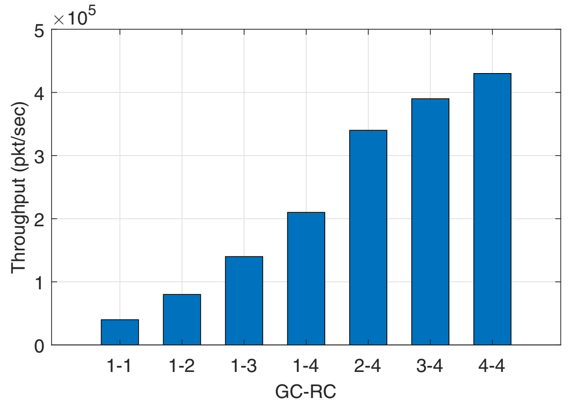
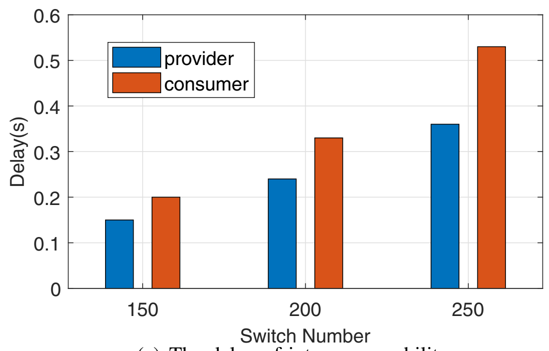


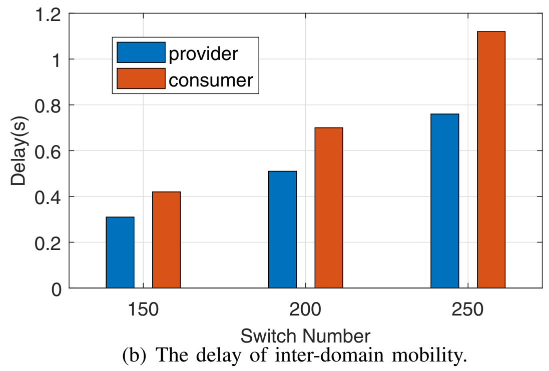
Fig. 9: Throughput of SPARC under different amount of GC and RC.

offloaded from root controller to nearby regional controllers, where interest requests are tried to be satisfied locally. This accelerates the processing speed of control plane. Second, when r = 4, the overall flow setup rate of SPARC rises significantly as the domain number ranges from 1 to 4. The reason behind this phenomenon is that more GCs can bring larger computation capacity. However, it can also be found that there is a slowing down of the rate growth, and it is caused by the east-west communication between GCs when handling inter-domain content delivery.

3) Mobility Evaluation: We conduct the third experiment to demonstrate the built-in mobility support of SPARC's ICN application. We employ four Linux server in this test, the number of switches/hosts N in each server is varying from 150 to 250, and the link number $M = 2N$. We let the variable x in N^x equals 1 and hence get 1,000 content and distribute these content to simulated hosts. A pair of end hosts is selected to act as content provider and consumer, then we use Pktgen to construct packets to randomly fake their mobility event. During this period, we measure the end-to-end delay variation by using TracesPlay [30] to analysis captured files. We first carry out the inter-area mobility test and the evaluation results are indicated in Fig. 10 (a). From the figure,we can see that the delay in publisher move scenario is more undulate than in subscriber move scenario, and the difference between the average value of these two scenarios is nearly above 0.1s. This phenomenon can be explain by the re-transmission of interest packet in the former case, which will introduce additional overhead. Moreover, we observe that when publisher moves, the delay time increases gradually with more switches are employed. This is also the case on subscriber move circumstance. It is easy to understand since the computational complexity grows when data plane scales to larger size. We also conduct the inter-domain mobility test and present results in Fig. 10 (b), and find the aforementioned phenomenons to be even more pronounced. For example, the average delay has at most 1.5× and 2.1× increasement respectively under the two moving scenarios when compared to the previous test. One important reason is that the inter-domain mobility process is



(a) The delay of inter-area mobility.



(b) The delay of inter-domain mobility.

Fig. 10: The delay of different mobility scenario.

more complicated and time-consuming as displayed in Fig. 6.

VI. Conclusion

In this paper, we design and implement SPARC, a scalable and programmable control plane architecture for translating high-level control program to large-scale POF network. SPARC enables local processing of network events by pushing down the intermediate representation closer to data plane for timely producing forwarding rules, and adopts a hybrid control plane architecture to improve the network scalability. Further, we realize a basic ICN routing and mobility module on the top of SPARC, and evaluate the performance of our system through this application. The experimental results show the feasibility and effectiveness of SPARC.

References

[1] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM Sigplan Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[2] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular sdn programming with pyretic," *Technical Reprot of USENIX*, 2013.

[3] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 217–230, 2012.

[4] C. Schlesinger, M. Greenberg, and D. Walker, "Concurrent netcore: From policies to pipelines," in *ACM SIGPLAN Notices*, vol. 49, no. 9, 2014, pp. 11–24.

[5] J. McClurg, H. Hojjat, N. Foster, and P. Černỳ, "Event-driven network programming," in *ACM SIGPLAN Notices*, vol. 51, no. 6, 2016, pp. 369–385.

[6] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[7] H. Li, C. Hu, P. Zhang, and L. Xie, "Modular sdn compiler design with intermediate representation," in *Proc. of the 2016 ACM SIGCOMM Conference*, 2016, pp. 587–588.

[8] M. Shahbaz and N. Feamster, "The case for an intermediate representation for programmable data planes," in *Proc. of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015, p. 3.

[9] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 103–115.

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

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

[12] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "Onos: towards an open, distributed sdn os," in *Proc. of the third workshop on Hot topics in software defined networking*, 2014, pp. 1–6.

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

[14] Y. Fu, J. Bi, K. Gao, Z. Chen, J. Wu, and B. Hao, "Orion: A hybrid hierarchical control plane of software-defined networking for large-scale networks," in *2014 IEEE 22nd International Conference on Network Protocols (ICNP)*, 2014, pp. 569–576.

[15] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "Rules placement problem in openflow networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1273–1286, 2015.

[16] X. Wang, Y. Tian, M. Zhao, M. Li, L. Mei, and X. Zhang, "Pnpl: Simplifying programming for protocol-oblivious sdn networks," *Computer Networks*, vol. 147, pp. 64–80, 2018.

[17] A. Abhashkumar, J. Lee, J. Tourrilhes, S. Banerjee, W. Wu, J.-M. Kang, and A. Akella, "P5: Policy-driven optimization of p4 pipeline," in *Proc. of the Symposium on SDN Research*, 2017, pp. 136–142.

[18] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[19] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proc. of HotSDN'13*, Hong Kong, China, Aug. 2013.

[20] "OpenDataPlane," https://www.opendataplane.org/, accessed: Feb. 14, 2019.

[21] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "Snap: Stateful network-wide abstractions for packet processing," in *Proc. of the 2016 ACM SIGCOMM Conference*, 2016, pp. 29–43.

[22] S. Vinoski, "Advanced message queuing protocol," *IEEE Internet Computing*, vol. 10, no. 6, 2006.

[23] Q.-Y. Zhang, X.-W. Wang, M. Huang, K.-Q. Li, and S. K. Das, "Software defined networking meets information centric networking: A survey," *IEEE Access*, vol. 6, pp. 39 547–39 563, 2018.

[24] L. Ding, J. Wang, Y. Sheng, and L. Wang, "A split architecture approach to terabyte-scale caching in a protocol-oblivious forwarding switch," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 1171–1184, 2017.

[25] R. Ravindran, A. Chakraborti, S. O. Amin, A. Azgin, and G. Wang, "5g-icn: Delivering icn services over 5g using network slicing," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 101–107, 2017.

[26] "Mininet," http://mininet.org/, accessed: Mar. 1, 2019.

[27] "PNPL prototype," https://gitlab.com/nhpcc416/PNPL, accessed: Mar. 3, 2019.

[28] "Floodlight," http://www.projectfloodlight.org/floodlight/, accessed: Mar. 3, 2019.

[29] "Pktgen," https://pktgen-dpdk.readthedocs.io/en/latest/, accessed: Mar. 3, 2019.

[30] "TracesPlay," http://tracesplay.sourceforge.net/, accessed: Mar. 3, 2019.