Facilitating ICN Deployment with an Extended OpenFlow Protocol

Piotr Zuraniewski[*]
TNO
Anna van Buerenplein 1
The Hague, The Netherlands 2509 JE
piotr.zuraniewski@tno.nl

Niels van Adrichem
TNO
Anna van Buerenplein 1
The Hague, The Netherlands 2509 JE
niels.vanadrichem@tno.nl

Daan Ravesteijn
TNO
Anna van Buerenplein 1
The Hague, The Netherlands 2509 JE
daan.ravesteijn@tno.nl

Wieger IJntema
TNO
Anna van Buerenplein 1
The Hague, The Netherlands 2509 JE
wieger.ijntema@tno.nl

Christos Papadopoulos
Colorado State University
1873 Campus Delivery
Fort Collins, Colorado 80523
christos@colostate.edu

Chengyu Fan
Colorado State University
1873 Campus Delivery
Fort Collins, Colorado 80523
chengyu.fan@colostate.edu

ABSTRACT

Named-Data Networking (NDN) is proposed as an approach to evolve the Internet infrastructure from a host- to an information-centric (ICN) approach, which is better suited to the current usage of the Internet. However, the deployment of a global NDN-based Internet is still a long way out of reach. The most likely scenario for a global NDN network will be the one based on NDN 'islands' or domains, where interior forwarding and routing of packets is based on NDN principles. The interconnection of NDN domains involves human configuration to set up IP tunnels, implying an unscalable, tedious and error-prone process resulting in static configuration incapable of reacting to ad-hoc requirements or network changes.

Leveraging the flexibility of Software-Defined Networking (SDN) can solve aforementioned problems. Due to its dynamic nature, SDN can automatically recognize an NDN service and instruct switches to set up the configuration for actual service deployment. Such a solution significantly eases the deployment of NDN networks.

In this paper, we propose a hybrid solution where we combine Software-Defined Networking, more specifically OpenFlow, and eBPF to perform control plane configuration and data plane programmability respectively, to realize connectivity within and across NDN domains. To do so, we have designed eBPF filters that match on NDN traffic, extended the OpenFlow protocol to configure switch data planes with these match filters and enhanced an OpenFlow switch to act accordingly. Our OpenFlow controller written for Ryu performs routing on NDN names and configures switches correspondingly. Additionally, our controller detects NDN domains and sets up IP tunnels between them. Our evaluation shows that our proof-of-concept on, among others, the SciNet testbed autoconfigures an NDN network, successfully providing end-to-end NDN network functionality across multiple domains.

[*]PZ is also affiliated with Department of Applied Mathematics, AGH University of Science and Technology, al. Mickiewicza 30, 30-059 Kraków, Poland

CCS CONCEPTS

• **Networks → Programming interfaces**; **Naming and addressing**; *Network performance evaluation*;

KEYWORDS

ICN Deployment, Softwarized Networks, Performance Evaluation

ACM Reference format:
Piotr Zuraniewski, Niels van Adrichem, Daan Ravesteijn, Wieger IJntema, Christos Papadopoulos, and Chengyu Fan. 2017. Facilitating ICN Deployment with an Extended OpenFlow Protocol. In *Proceedings of ICN '17, Berlin, Germany, September 26–28, 2017,* 11 pages.
DOI: 10.1145/3125719.3125729

1 INTRODUCTION

Undoubtedly, the research on various Information-Centric Networking (ICN) designs, most notable Named-Data Networking (NDN, [5]) and Content-Centric Networking (CCN, [11]), is ramping up. However, the deployment of a global ICN-based Internet, where an ICN-based protocol takes the networking forwarding role currently occupied by IP, is still a long way off. For the foreseeable future, the most likely scenario for a global ICN network will be the one based on ICN 'islands' or domains, i.e. smaller networks where internal forwarding and routing of packets is based on ICN principles. Connectivity between such 'islands' will be ensured via tunnels over the IP-based Internet. Some early examples of these type of deployments are the SciNet testbed [13] and NDN Community Testbed [5], where several dozens of NDN domains in total, hosted at research institutes around the world, are interconnected to form connected NDN networks.

The interconnection of ICN domains currently involves human intervention to set up IP-encapsulating tunnels, which in the long run implies a tedious and error-prone process that does not scale. In fact, human involvement is typically needed to set up any non-trivial ICN service. In practice, this means that the configuration

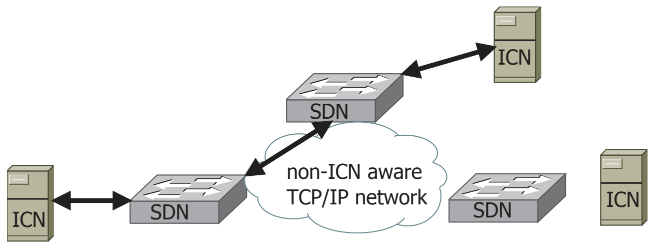

Figure 1: Tunneling idea illustration

of such a service is relatively static and not based on ad-hoc requirements or topology dynamics. There are currently no commonly used mechanisms to instantiate an ICN network with, e.g., certain QoS characteristics on demand. The absence of 'ICN-as-a-service' hinders the widespread deployment of ICN on a global scale. Though we do not think that a global adoption of ICN will happen very soon (cf. IPv6 casus), we believe that this process can be significantly sped-up and facilitated.

We envision that leveraging the power and flexibility of Software-Defined Networking (SDN, [19]) can help in combating the aforementioned ICN deployment problems. Due to its dynamic nature, SDN can automatically recognize a request for a given ICN-related service by analyzing the incoming packets, possibly working hand-in-hand with native ICN protocols such as a routing protocol. The SDN controller can set up flows and tunnels on the 'gateway' SDN switches (i.e., the switches at the border of ICN and SDN domain, see Figure 1) over non-ICN-aware TCP/IP networks to achieve service deployment and perform garbage collection when a service is not needed any more. Such a solution significantly increases the ease of deployment of ICN networks.

However, OpenFlow-based SDN (OF, [21]) does not allow for easy evaluation and deployment of new protocols out-of-the-box. The reason for this is that the OpenFlow standard defines only a limited set of datagram header fields (MAC, IP, TCP/UDP port, etc.) on which SDN switches can match and therefore base their forwarding decisions on. While with each version of OpenFlow the number of these fields increases, new standardized protocols arrive in much higher pace. In theory, it is possible to make an own branch of OF and add support for a specific field of a specific protocol, however, such a solution is not feasible as it would create myriads of incompatible versions of the OF protocol that would not be compliant with the standard. An alternative workaround, i.e., sending all the datagrams to the controller where an application capable to parse them would run is infeasible due to the unscalability of a controller processing each packet to be forwarded.

A fundamental difficulty in using SDN to support ICN is therefore processing the ICN packets by an SDN switch: without understanding the syntax and semantics of ICN datagrams, no action can be taken by the underlying network. Unfortunately, OpenFlow does not offer any support for the various ICN packet formats currently being discussed. Even if ICN support would be added to OpenFlow, any change in ICN specification would require new amendments to OF as mentioned above. Moreover, the structure of the packet itself in, for example, the case of the NDN protocol is very complex. We face a nested Type-Length-Value structure (i.e., inside a TLV we can have further TLVs) with a variable size of not only Value but Type and Length components as well. Due to its format's complexity, this use case is a perfect illustration of the problems that can be solved with our proposed solution: we deal with a protocol currently unsupported by OF, with a highly nontrivial header structure, being subject to rapid evolution.

In this paper, we introduce a mechanism (inspired by the framework outlined in [17]) that overcomes the aforementioned limitations and allows SDN switches to forward packets based on arbitrary fields of any protocol header. Using the extensibility features of OpenFlow in combination with Extended Berkeley Packet Filters (eBPF, [28]), we enable intelligent forwarding of packets without having to go through a central SDN controller for every datagram of the currently unsupported protocol. Also, currently unsupported fields of the well-supported protocols (f.e., TCP flags in OF version 1.3) can be handled precisely in the same way. Moreover, while our current demonstrator implementation is fully software-based, the very recent developments in network interface cards promise to allow (e)BPF program execution in hardware [2] using Linux *tc* or XDP[18], opening future possibility for a line-rate performance of the discussed method. Our proposed framework is therefore OF standard complaint, future-proof and allows production deployment.

The contribution of this paper is the following: we have extended the approach proposed in [17] and created a framework that allows for easy development of parameterizable, flexible eBPF programs being capable to match on an arbitrary part of a datagram of virtually any protocol, even as complex as ICN. We did so by extending the OpenFlow protocol with our own vendor extension to enable the switch to match packets based on the output of parametrized eBPF programs. We have also extended the SDN controller to be able to transport possibly complex and large eBPF matching programs and, finally, we have modified the SDN OFSoftswitch [4] to be able to use the introduced OpenFlow extensions and execute the compiled eBPF program. To assure reproducibility of our research, we release the source code used for developing our proof-of-concept [29][30].

The rest of the paper is organized as follows: In Section 2, we discuss the related work. A brief description of the most important features of the technologies involved, i.e. SDN, ICN and eBPF can be found in Section 3. In Section 4 we provide the details of the proposed solution describing how flexible matching is performed and what extensions to OpenFlow and the controller were made. We experimentally evaluate our proof-of-concept in Section 5. Finally, section 6 concludes the paper and indicates future research directions.

2 RELATED WORK

One could argue that the issues discussed above could be resolved using *packet_in* SDN mechanism. In such a system, every incoming ICN packet is sent to the SDN controller, which, assuming it is ICN-aware, can then compute the path to the final destination for that packet. There are a number of disadvantages to this approach, one of which is that the central controller can potentially be overloaded with a large number of packets to process as each and every packet of the ICN flow would have to go through the controller. Additionally, some extra delay will be introduced due to the switch-controller communication.

In [32], the authors describe NDNFlow. In this solution, a second communication channel, parallel to OpenFlow, is proposed that can be used to communicate with an ICN-aware module in the SDN controller. When an ICN-aware SDN switch receives an Interest message, it communicates this information to the ICN controller module which then inserts flows in both ICN-aware and non-ICN-aware SDN switches along the path to the final destination as determined by the ICN controller module, thereby creating an ad-hoc IP tunnel for the ICN packet to flow through. While this mechanism is more efficient than sending each and every ICN packet to the SDN controller, in the current implementation it results in 'static' paths that are only valid for specific ICN names (note, however, that introduction of some more dynamism and parametrization appears to be possible). Further difference is in dealing with the variable length TLV fields used in current NDN and CCN implementations: in NDNFlow this functionality is offloaded to CCNx daemon which is expected to create an extra overhead due to communication as compared to our solution where all the processing is done locally on a switch. Moreover, as discussed earlier, full hardware acceleration of eBPF programs seems to be possible in near future while it is unclear if NDNFlow can also enjoy it. Finally, NDNFlow deals only with NDN: eBPF-based solution can be very easily adapted to handle any other protocol.

In [33] and [22] the authors propose a framework for extensions to OpenFlow that would allow various degrees of ICN awareness for SDN-switches, ranging from being able to forward ICN packets to being able to perform more complex ICN functionality such as caching. While the proposed mechanism is very suitable for the long-term evolution of SDN to make it compatible with ICN and other future network architectures, the proposed mechanism would require significant changes to how OpenFlow currently works.

Our proposal is to extend the method for arbitrary packet matching in OpenFlow based on (legacy) Berkeley Packet Filters (BPF) first described in [17]. The authors introduce a system that uses OpenFlow Extensible Match (OXM) to transfer a compiled BPF program to an SDN switch which can then match on the result returned by the BPF program.

In our proposed solution, we borrow the idea of using BPF in OpenFlow, however, we introduce several modifications which address some of the shortcomings of the approach outlined in [17]. The first limitation stems from using OXM to store BPF programs: it does not allow to have bytecode longer than 248B. In an ICN context, this is a very heavy limitation because we not only have to store a (possibly long) ICN name itself but also the whole matching program which – due to the complexity of an ICN packet and the fact that the BPF bytecode only supports forward jumps – can easily take several kilobytes. While the authors of [17] suggest that this difficulty may be circumvented by fragmenting the BPF program, which would require extra book-keeping, padding etc. Instead, we propose to use the OpenFlow experimenter messages [7, 7.5.4] to carry the program. The maximum size of these experimenter messages can be as high as 64kB, leaving plenty of space for matching to any parameter in an ICN packet, if needed. We also use a new flow match field for invoking the BPF program per packet for the matching itself.

Moreover, the design discussed in [17] does not mention the possibility of passing arbitrary parameters to the BPF program. We have incorporated parametrization, which means that in an ICN context it is sufficient to have just one program for every ICN name we want to match on (i.e., a name is a parameter) instead of compiling and shipping a new program each time the desired match string (e.g. ICN name) changes.

Last but not least, we propose to use Extended BPF (eBPF, see [20]) instead of legacy (classic) BPF as in [17]. From a usability perspective, this offers a possibility to develop matching programs in (restricted) C instead of BPF assembly, making the development process much easier. Additionally, just-in-time (JIT) compilation for eBPF is available for 64-bit x86 architectures increasing program execution speed and eBPF is also supported in the Linux kernel.

Arguably, our particular tunneling use case picked to demonstrate the concept can partially be solved using other methods, such as NLSR [14]. However, our solution is not intended for the single purpose of replacing NLSR. It is a method to dynamically match on arbitrary packets, including ICN/NDN, and take arbitrary actions on those, hence, easing the complexity of deploying new forwarding techniques in existing programmable networks. Additionally, our routing concept, described in more details in Section 4, can also take into account intermediate ICN caches placed mid-tunnel. Whereas with NLSR, the tunnels are predefined and due to the encapsulation of IP, packets cannot break out of them mid-path.

To summarize, our proposed framework allows for much easier development of parameterizable, more flexible and advanced eBPF matching programs as compared to [17]. At the same time, because eBPF allows packet matching to be performed on the switch itself instead of having to go through a central controller, the switch-controller traffic volume is significantly reduced compared to [32], and so is the delay associated with it.

P4 [8], [27] is a high-level language for programmable data planes. P4 allows developers to redefine packet processing control-flow. While P4 is certainly a very interesting data-programmability tool, in our approach, we adopt eBPF as the engine for programmable data planes for several reasons. First of all, users have different capabilities when using P4 and eBPF. To handle significant changes such as the introduction of new operations, merely a modification of eBFP program is needed. With P4, users may need to wait for the new P4 version to be released (for now, the transition to 16) which has to take a long time to be accommodated by the rest of the ecosystem. The difficulties and restrictions encountered when parsing NDN packets (or in general TLV-structured packets) using P4 language are discussed in depth in [26]. We have not encountered them in our work, at the minor expense of using slightly less abstract programming language (restricted C). Second, P4 targets compile-time data plane agility, whereas eBPF is run-time data plane agility. When the network ecosystem changes due to for example packet header format change, both P4 and eBPF program needs modifications accordingly. Since the P4 language provides an abstraction to compile an entire switch from scratch for a specific target (FPGA, NPU, x86 etc.) it is then still required to adapt or implement the P4 generated control API to the specific control application. For eBPF framework merely a change in the matching program itself is needed while the rest of the components (a switch, a controller) remain the same. Last but not least, regarding hardware acceleration (which is a hallmark for P4) it is worth to notice that the very recent patches into the Linux Kernel already support initial hardware accelerated eBPF programs for specific network cards [2] although, at the time of writing, full support for all eBPF functionality is not yet there. We discuss hardware acceleration possibilities with eBPF in more details in Section 6.

3 ICN, SDN AND EBPF IN A NUTSHELL

In this Section we will give a brief description of the ICN features which pose a challenge for widespread deployment from the network infrastructure perspective, of SDN and eBPF and how SDN (coupled with modern tools like eBPF) can act as a facilitator.

3.1 ICN

Two of the implementations of ICN at the time of writing are NDN [5] and CCN [11]. In NDN, each packet is encoded in Type-Length-Value (TLV) format. NDN Interest and Data packets do not have a fixed header but are essentially a collection of nested TLVs, where some TLVs can contain sub-TLVs, which in turn may be further nested. The first and outermost TLV defines whether a given packet is an Interest or Data packet. To reduce the total packet size and to allow for future extensibility, the Type and Length of a TLV are encoded with a variable length. This makes it impossible to know the position of an NDN name field *a priori* and involved processing steps have to be taken to decode the full ICN name from an NDN packet.

In CCN, the packet format is slightly different and makes it easier to process since it has fixed length Type and Length fields. However, given that the name of an Interest or Data message contains, by its very definition, a variable number of name components, it is still not possible for a switch to match on a fixed set of bytes in the CCN packet and to base its forwarding decision on, e.g., one or more *Name* components.

3.2 SDN

Separation of the control and data plane is most frequently used as a short description of what SDN is. It means that the switching elements in the network do not have much intelligence but rather forward the traffic, relying on the instructions obtained from the (logically) centralized controller. OpenFlow is the most widely used protocol to assure communication between the switches and controller.

One of the visions behind SDN is that network administrators do not have to wait until some features or protocols are implemented by the vendors. Instead, a desired functionality can be developed in-house and deployed just like it happens with new applications running on computer operating systems. In practice, however, this process can be challenging as we have witnessed with ICN due to lack of support for ICN protocol suites in OpenFlow. Nevertheless, thanks to the fact that a vast majority of the code and interfaces in the SDN realm are open, as we demonstrate in our paper, many of the obstacles can be removed. It proves that SDN can be regarded as an enabler for the other communication technologies.

3.3 eBPF

Berkeley Packet Filter was introduced in [20]. It is used for example in the popular capturing tool *tcpdump* where the user can define which packets should be captured (e.g., only UDP or TCP with a source port larger than 1023). A condition (or set of conditions combined using logical operands) is then compiled into bytecode which is in turn executed allowing for fast packet filtering. In a general case to use BPF, an assembler program has to be developed by the programmer. After compilation, the program is executed by a virtual machine (VM) with optional JIT compilation. A notable feature of this VM is that only forward conditional jumps are allowed, which guarantees that BPF programs will end, making it ideal for insertion in critical paths (such as in an operating system kernel)

While developing simple programs is relatively easy, it becomes a daunting task if a large number of conditions must be checked due to, for example, complexity of a packet structure, which is exactly the case for ICN. Recently, a modernized version of BPF was introduced, known as Extended BPF or eBPF [28]. From a usability perspective, one of the most prominent changes is that the programs can now be developed in restricted C, greatly simplifying the development process.

4 IMPLEMENTATION DETAILS

The OpenFlow (OF) protocol does not natively support matching on an arbitrary part of the datagram. In the ICN context it means that we cannot easily instruct a switch to match on, e.g., the second *NameComponent* field of an interest packet – the position and length of this component are not known *a priori* – rather this information is encoded in the datagram itself. Rather, in the OF specification there is a number of pre-defined fields such as ingress port, MAC address, IP address, UDP port, etc., etc. on which flow rules can match. With each version of OpenFlow the set of match fields is expanded, however, even in the newest proposed version (OF 1.5) this is limited to 45 predefined fields. Moreover, to make a switch compliant with a given version of OF, a switch has to support only a subset of match fields. In OF 1.3 [7] (currently considered stable and one of the mostly deployed versions) the set consists of 13 fields only.

In view of this and the fact that new essential specifications of OF are released roughly every year, it is difficult to quickly introduce handling of new protocols (such as the ones from the ICN suite) to the SDN network and stay compliant with the OF specification. Fortunately, OpenFlow also defines (an optional) experimenter flow match field [7, 7.2.3.10] along with an experimenter instruction [7, 7.2.4], i.e., an instruction which is executed after a flow matches the entry. There is also an experimenter message defined [7, 7.5.4] to carry additional information between switch and controller that is not handled by the standardized messages. These three elements do not – by themselves – solve the problem of flexible matching; though they offer a framework to define additional functionality, not covered by the base specification. As we found out in our research, several other important components were needed to realize arbitrary matching; we will describe them now in more details.

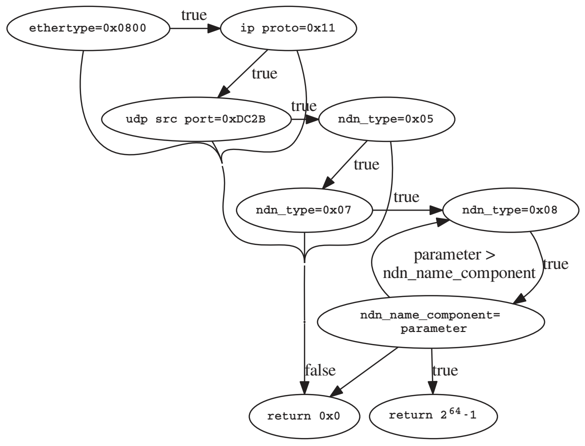

Figure 2: Parse graph for NDN interests. Unlabeled arrows mean 'false' condition and merging arrows indicate a merge of paths towards a given action.

4.1 Matching engine

We have used the eBPF framework (see Section 3.3) to develop a matching program. The proposed solution allows for matching using complex conditions on a packet with a non-trivial structure and is in principle not limited to ICN. Nevertheless, ICN (and especially NDN) packets, due to their complexity, are good candidates to demonstrate capabilities of the proposed solution.

Typically, a switch matches on specified fields extracted from the packet header (e.g., destination IP address) or some meta-data associated with the packet (e.g., ingress port number). In our situation, a match is performed on the value returned after execution of our eBPF program which takes the packet (along with possible meta-data) as an argument. To extend flexibility, we provide an additional meta-data field which can be used to parametrize the eBPF program, e.g., in the case of ICN, this could be the NDN name, see Listing 1. The program returns a 64b value on which masking and the actual flow matching are performed by the OF switch.

Listing 1: Structure for matching

```
struct ofsoft_bpf {
  //OFSoftswitch13 metadata
  uint8_t table_id;   uint32_t in_port;
  // Match parameter
  uint8_t* param;     uint8_t param_len;
  // Packet
  uint8_t* packet;    size_t packet_len;};
```

4.2 Setup

In our implementation we have a created a testbed consisting of a mix of readily available software as well as some newly-developed and modified components. Our ICN stack of choice was NDN's NFD v. 0.4.1 [5]. No custom modifications to the codebase were introduced. UDP multicast faces were used for host-to-host communication. Note that it is not mandatory to use these particular faces, however, they constitute a straightforward option for our setup since it enables us to dynamically add nodes to our testbed with minimal per-machine configuration. Regarding a switch, we have started with the CPqD Open Flow 1.3 Software Switch (OFSoftswitch) [4] modified as described in [17], which we have then further enhanced with our own vendor extension and logic to realize matching on eBPF programs [29]. For the OpenFlow controller we have picked Ryu [10], starting with the previously modified version [17] and introduced functionality handling eBPF programs and ICN routing [30]. Finally, a userspace eBPF virtual machine known as uBPF was used [12] along with clang-3.7 compiler [3]. See also Figure 5 in Section 5 for a sample topology and separation of components.

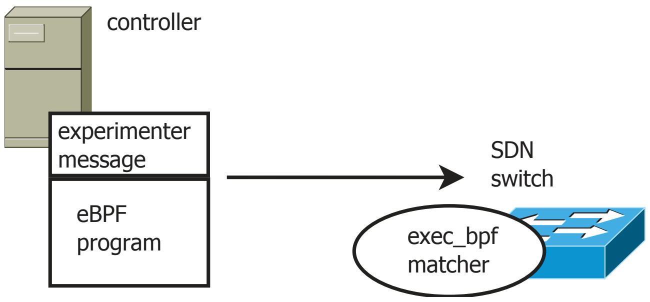

Figure 3: Overall flexible matching architecture

4.3 eBPF ICN matcher

In this Section we will describe in more details the logic behind the ICN matching program. Its basic user-specified parameter is a string that must be present in the ICN *Name* field to have a match. The 2nd parameter of the program is a byte-encoded ICN packet along with meta-data such as the ingress port and table ID to be matched against.

As mentioned in Section 4.2 we have decided to use a UDP face for communication between ICN nodes running an NDN stack. Therefore, the program first performs a series of checks on ethertype, the protocol carried by the IP packet and UDP port number. Next, after verifying that a datagram is an NDN *Interest* packet, its nested TLV *NameComponent* fields are searched (see Figure 2) by the loop where the *parameter*, constituting the name we are looking for, is compared to the subsequent *ndn_name_component* field. Ultimately, the program returns a value of 0 (string not present in the Name field) or $2^{64} - 1$ (string present in the Name field). This value is then used to decide which action will be taken by a switch. The matching program is compiled exactly once (no recompilation is needed due to parametrization of the names) and is ready to be handed over to the controller.

4.4 Implemented OpenFlow extensions

In this Section we will describe the modifications which were needed to achieve flexible matching. Refer to Figure 3 for an overall architecture.

- We extended the OpenFlow match fields with our own *exec_bpf* flow match field. Our introduced match consists of 4 fields: i) the program number which is the 32b identifier associated with an eBPF program in our vendor extension, ii) a value of 64b to be compared with, iii) a 64b mask, and

Table 1: Forwarding actions performed on ICN packets

Node	Action	MAC-src	MAC-dst	IP-src	IP-dst	UDP-port (optional)
icn01	Locally broadcast Interest	icn01	multicast	n.a. / icn01	n.a. / multicast	n.a. / 56363
sw01	Rewrite MAC-dst, add / rewrite IP+UDP header		gateway	icn01	icn02	6363 (dst)
IP network	Regular IP forwarding	gateway	icn02			
sw01	Forward					
icn02	Construct and reply ContentObject, inverse src / dst	icn02	gateway	icn02	icn01	6363 (src)
sw02	Forward					
IP network	Regular IP forwarding	gateway	icn01			
sw01	Forward					
icn01	Receive ContentObject					

iv) a variable parameter field with a maximum length of 220 bytes. After execution of the eBPF program, the 64b result returned by the eBPF program is masked with the user-defined mask and compared to the value defined in the flow. While in our case only a 1-bit result is used (either a match or not), we keep options open for future enhancements and maintain compatibility with TCAM style matching and result returning. Upon a successful match the associated actions are executed. The simplest action is forwarding to a given output port but OFSoftswitch also defines more advanced actions such as pushing an MPLS label or moving to the next table with another set of rules. In case that no match is made, the normal OpenFlow process continues.

- The Ryu controller did not offer any means to handle (e)BPF programs. We therefore modified it by adding support for the introduced match field *exec_bpf* and added a vendor extension to transmit the eBPF program to OFSoftswitch. Specifically, we introduced two operations:
  i) *putBPF* containing the compiled eBPF program and its 32b numeric identifier, enabling loading multiple eBPF programs on OFSoftswitch; ii) *delBPF* containing the 32b identifier of the eBPF program to be removed.
  For both *putBPF* and *delBPF*, OF experimenter messages must be used since such operations cannot be handled by any of the basic messages defined in the OF standard. A single experimenter message can have a length of 64kB, allowing for large programs to be sent. When 64kB is not enough, multipart experimenter messages can be used [7, 7.3.5.15]. For reference, our program, which has not been in any way optimized to use as little space as possible, has a size of 3kB after compilation.
- OFSoftswitch was modified to work with our OF extensions. The data plane was extended with uBPF – eBPF VM developed by BigSwitch Networks [12].

4.5 Routing and Forwarding

To aid in routing, we adopted and extended the multi-switch forwarding controller app for Ryu presented in [31]. In particular, we extended the app to support disconnected layer-2 broadcast domains by isolating layer-2 traffic between them and implemented the functionality to share the app's adjacency and forwarding tables with the ICN and eBPF part of our work. While forwarding between the layer-2 network domains works automatically through the preconfiguration of IP gateways on the ICN nodes – the MAC layer is oblivious to the layer-3 forwarding though does configure the layer-2 rules necessary for an ICN node to forward packets to the router's MAC address – ICN forwarding rules and eBPF match filter configuration are computed and installed on the switch by a separate ICN app.

While the forwarding app takes care of traditional Ethernet and IP forwarding of traffic, the ICN app handles all ICN routing and configuration. To do so, it maintains a table containing mappings from name prefixes to particular hosts. To fill this table, the ICN nodes run a simple routing protocol, effectively propagating their ICN capabilities and name prefixes to the controller through the JSON schema displayed in Listing 2. Initially, all switches have no ICN configuration present, implying they forward incoming Interest messages to the controller for further inspection through OpenFlow's *PacketIn* message (typically, it happens only for the first packet in the flow). The ICN app selects the most suitable available location to set up a route to by selecting the name prefixes that have the most matching prefixes (a.k.a. longest-prefix match). Additionally, it can sort on the priority field and where applicable the path costs summed with the generator's initial cost metric to aid in routing decisions.

Listing 2: JSON encapsulation of NDN routing information the node's MAC and IP address are derived from the headers surrounding this message.

```
[
        "name" : <prefix-name>,
        "priority" : <strict priority>,
        "cost" : <initial cost metric>
]
```

Where the ICN node performs a local multicast of the packet – it is oblivious to the routing decisions made by the network – our switch replaces the multicast MAC and IP addresses with the unicast addresses[1] of the next ICN hop on the path, where necessary replacing the MAC address with the MAC address of the local IP gateway. Using this method, we effectively set up an IP tunnel to the next hop on the path. Alike to regular gateway configuration, we have used a static table containing the locations and addresses

[1]Additionally, we increase the IP Time-to-live and UDP port number values to reflect this.

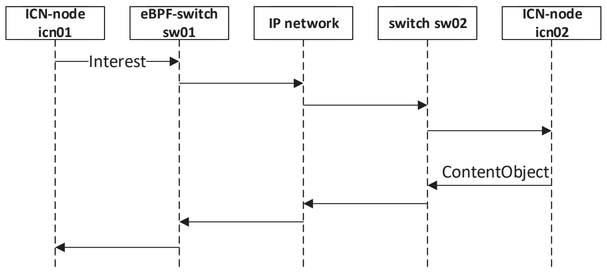

Figure 4: Packet traversal across nodes, see Table 1 for the respective actions per node

of the local routers. Recursively, the ICN app uses the MAC learning table and adjacency and routing tables from the forwarding app to fill in the exact fields of the rules it generates. Using this method, we forward *Interests* along a path of short-tunnels between NDN nodes, ultimately to the generator of the data required. Note that although not the whole network is Software-Defined, nor under our administration, we are able to dynamically set up routing between any Software-Defined Network under our administration as long as they are somehow interconnected through traditional IP gateways. In our Proof-of-Concept we have limited ourselves to setting up IP-encapsulating tunnels, however, from a routing perspective our solution is protocol-agnostic and may be mixed with other tunneling and labeling techniques such as MPLS or GRE to connect otherwise disconnected NDN islands. Figure 4 and Table 1 show the various packet actions applied by our solution.

Where fully-implemented NDN stacks employ the state of the Pending Interest Table to forward *ContentObjects* back to their original requester, it is difficult to store local state using eBPF or P4 programs. To the recipients of the Interests, however, the packets seem to originate from the unicast address of the source node and will reply likewise. Hence, to guarantee returning *ContentObjects* are delivered to their source in a proper way, we only need to add regular MAC and IP forwarding rules on the intermediate switches. Through this method, the returning ContentObject follows the reverse-path of ICN nodes visited by the originating Interest, guaranteeing both optimal operation and execution according to the NDN specification. In particular, all intermediate nodes can cache the resulting ContentObject according to the NDN specification. Additionally, our routing scheme can be enhanced to configure more than one forwarding rule for each propagated name-prefix to exploit NDN's Strategy Layer.

5 EXPERIMENTAL EVALUATION

To evaluate and verify our proposal, we deployed our tools on the SciNet testbed [13]. The SciNet testbed contains seven nodes that are located at institutes such as Colorado State University, Lawrence Berkeley National Lab, NCAR-Wyoming Supercomputing Center and CalTech. These nodes are connected through 10 Gbps links. Large volumes of data are transferred through NFD daemons and other scientific applications. In addition to the high-speed links, the testbed can also dynamically create end-to-end, bandwidth-reserved paths on-demand between the testbed machines using ESNet's circuit reservation system, OSCARS. To access additional scientific datasets (such as climate and high-energy physics datasets), the SciNet operators are expanding the testbed to cover new locations. Currently, operators have to manually configure the channels between nodes to enable the interconnection of NDN nodes, which is a tedious and error-prone process that does not scale.

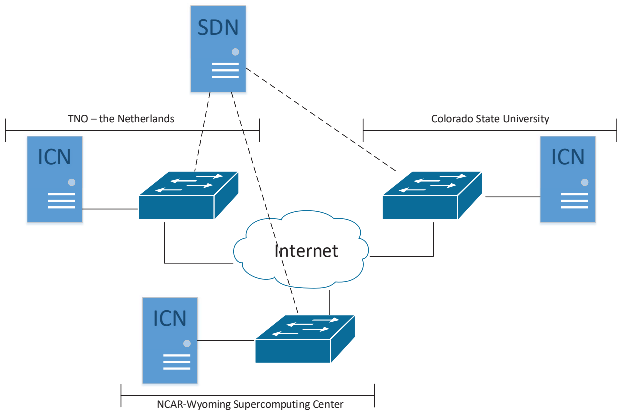

Figure 5: Overview of testbed hosted at SciNet and TNO

From SciNet, we have connected the Colorado and Wyoming locations to the TNO NDN testbed in the Netherlands through regular IP-based Internet as presented in Figure 5. By interconnecting the initially separated ICN domains through regular IP, we ultimately show that our solution also works in the extreme case where the interconnecting medium and technology is not under the administration of the interconnecting parties.

Section 5.1 discusses our experiences running NDN on the SciNet testbed. In Section 5.2, we evaluate the switch performance of our proposal.

5.1 Experiments on global testbed

To show that we achieve our goal of forwarding NDN packets using our proposal, we have deployed our implementations on the SciNet testbed connected to the TNO NDN testbed. Our experiment contains two nodes hosted at SciNet (USA) and one hosted in the Netherlands at the TNO testbed (Figure 5). First, we verify connectivity and perform latency measurements using *ndnping* from NDN Essential Tools [23]. Second, we achieve and measure file transfers using *repo-ng* [25]. The 3 different testbed locations all host an unmodified ICN node and a modified SDN-based switch according to Section 4.2. The switches connect to our modified SDN controller running at TNO location.

Overall, connectivity between the 3 locations is automatically configured by the SDN controller. The SDN switches forward the first packet of new flows between each other to the controller, which then installs the corresponding eBPF program and parameters to

Table 2: Experiment 1 – maximal loss-less packets-per-second (PPS) values. Means and standard deviations calculated over 30 **runs.**

test set-up	eBPF match	IP/MAC re-write	last loss-less [PPS]	first loss [PPS]	last loss-less mean [P]	first loss mean [P]	first loss sd [P]
(A)	Y	Y	2100	2200	100000	99997.8	9.48829
(B)	N	Y	2100	2200	100000	99999.9	0.50742
(C)	Y	N	4000	4100	100000	99998.1	10.5893
(D)	N	N	4100	4200	100000	99998.4	8.94614

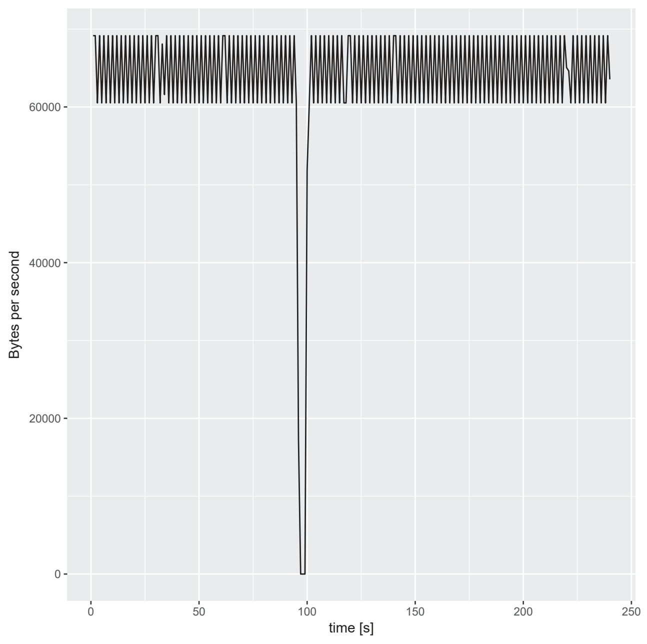

Figure 6: Throughput of file transfer between Colorado and the Netherlands

forward that flow through our extended OpenFlow protocol. Figure 6 shows the throughput of a, single though representative, file transfer between the locations in Colorado and the Netherlands, performing at an average of 63, 841.5 B/s with a standard deviation of 8, 944.9 B/s and 44.3 packets/s with standard deviation of 6.2 packets/s. The NDN file transfer application used does not employ congestion windows or flow control (this is typical for current NDN deployments). The frame rate achieved is well below switch capabilities (see Section 5.2 below) which means the bottleneck here is an application itself, not our modified switch. In fact, the sample file transfer application works in such a way that a next interest is sent only after data satisfying a previous interest is received. For the networks with large delays it has immediate consequences in terms of throughput. Delay measurements through *ndnping* and *repo-ng* both showed an RTT of around 130 ms. Given that ContentObjects with a size of 8, 500 bytes were sent, our measurements verify the theoretic throughput of 8, 500 B / 130 ms $\approx$ 65, 000 B/s. Using a different application offering better flow control, such as *chunks* [24], is expected to further increase performance of the file transfer in practice. Hence, we conclude that our proposal can dynamically set up successful connectivity between NDN-islands while still providing the level of service expected from static configuration. Furthermore, no modification of any element of our demonstrator was needed to connect *repo-ng* application to the running experiment which proves flexibility of our solution.

5.2 Performance evaluation

Additional to the previously presented testbed evaluation, we have evaluated the forwarding performance of the enhanced software-based OpenFlow switch. To do so, we have installed the modified switch on a four-core virtual machine running Ubuntu 14.04LTS dedicated to this evaluation and monitored its behavior under varying network loads. The packets reaching the switch were generated on a separate VM. The switch VM and packet generating VM were hosted on a commodity server (dual Intel E5-2683v4, 256GB RAM) running Ubuntu 16.04LTS using the KVM hypervisor. The priority of the switch application (which operates in user space) was increased by setting its *nice* parameter to -10. To maximally isolate an environment for the experiment, we prevented the replies (data packets) to reach the switch. The returned data objects would also consume CPU time on the switch without actually running eBPF matches and hence unjustifiably and adversely impacting results. Furthermore, we used separate VMs from the global testbed ones used in Section 5.1.

Each experiment was executed in four set-ups:

(A) operational, when eBPF matching and packet header manipulation operations (IP/MAC rewriting etc.) were both in effect;
(B) non-operational, without eBPF matching but with header manipulation;
(C) non-operational, with eBPF matching but without header manipulation;
(D) non-operational, without eBPF matching and without header manipulation.

The non-operational set-ups are executed to assess how expensive the individual eBPF and rewriting parts are, with setup (D) being a baseline scenario since no operations except for plain packet forwarding are performed. For the set-ups with eBPF matching, before starting the actual experiments we sent one interest packet to assure that a flow is installed on a switch. This is to prevent any packets being lost due to a transient state when the controller and switch are in the process of computing and configuring the flows.

For each set-up, we have performed the following experiments:

- Experiment 1: Maximal packets-per-second (PPS) value for which no packets are lost.

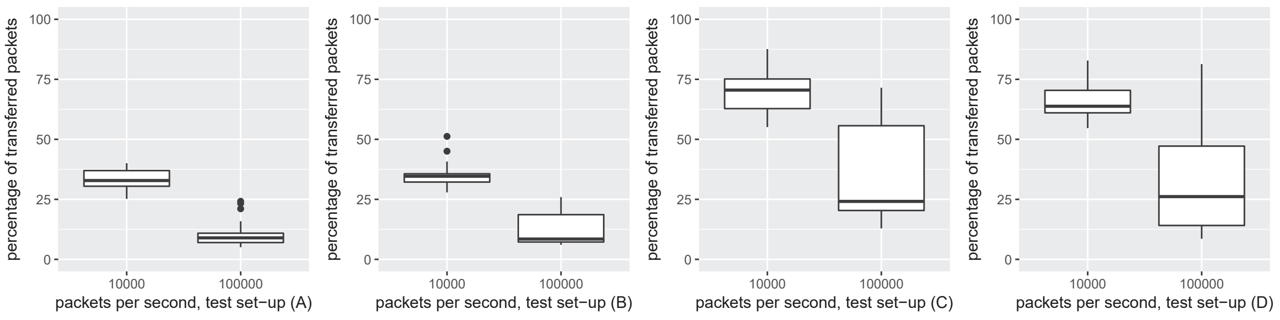

Figure 7: Experiment 2 – packet storm

We performed a grid search for packet-per-second values with steps of 100 looking for the highest PPS value for which we see no packet drops as they traverse a switch ('last loss-less') and the first PPS value for which packet drops are observed ('first loss'). We sent a stream of NDN interest frames towards our modified switch using *tcprelay* at a pre-set rate. For each speed, thirty repetitions were done using the same *pcap* file with 100, 000 Interest packets. Mean values and standard deviations for the interesting quantities were calculated. The results are gathered in Table 2. From these results, we conclude that, in fact, it is packet header manipulation, not eBPF execution, that is the most expensive operation. We see that the highest found loss-less transfer rate is of order of 4000 PPS when header rewriting is absent (setups (C) and (D)) and is of order of 2000 PPS when header rewriting is present (setups (A) and (B)). The 'cost' of header manipulation is therefore roughly 2000 PPS.

- Experiment 2: Packet storm.

  We have also verified the performance where the PPS rate is well beyond the values found as stable in Table 2. We have picked values of 10, 000 and 100, 000 PPS, conducting the experiments for all set-ups (A) – (D) as described above. We replayed the same *pcap* file with 100, 000 interest packets and performed 30 repetitions of each experiment. The results are gathered in Figure 7. Visual inspection of the box-plots shows that the distributions of the percentage of packets which were successfully transported over the switch differ.

  More formal tests like two-sample Kolmogorov-Smirnov gives p-values in the order of $O(n^{-8})$ – $O(n^{-16})$, which means rejecting the hypothesis of the distributions being equal at any reasonable significance level. In brief, as it could be expected, the more burden is put on the switch the smaller the chances are for a packet to be transferred. Furthermore, the conclusion from the previous experiment that packet rewriting is the most expensive operation holds if we compare set-ups (A) and (B) - where packet rewriting is present - to set-ups (C) and (D) - where packet rewriting is absent. Regardless of the PPS value, set-ups (A) and (B) yield smaller percentages of transferred packets compared to set-ups (C) and (D).

- Experiment 3: Constant PPS with variable packet size.

  In this final experiment, we have fixed the PPS value being the found "last loss-less value" for each set-up, respectively being 2100, 2100, 4000 and 4100 for set-ups (A) – (D) as found in Table 2. The introduced variable in this experiment is an increasing packet size. To implement increasing Interest packet sizes, we have increased the number of name components present in the Interest. To prevent interference of matching longer names on the forwarding speed, matching of the NDN name - as applicable in cases (A) and (C) - is only performed on the first name component.

  The results are gathered in Table 3 and Figure 8 (no error bars are plotted since they are too small to be visible). For the larger packet sizes, the switch starts to drop more packets. This is expected behavior since a bigger packet implies more CPU cycles to be processed. Another observation is that the values which were considered as 'last loss-less values' as the result of 30 repetitions in experiment 1 (see Table 3) did not always yield 100% transmitted packets (packet sizes of 78B, set-ups (A) and (C)). At the same time, for a larger packet size where some losses could be expected we see all the packets being transferred (packet size of 132B, test set-ups (A) and (B)). We attribute this behavior to the fact that the switch application runs on a virtual machine, fully in user space. As a consequence, the influence of e.g., a scheduler running both on the host and guest cannot be neglected. Nevertheless, the differences were not large and we decided to maintain the values found in Table 3 with the remark that some of them may be slightly too optimistic or pessimistic.

The performance evaluation experiments give us a baseline result for further research and improvements. While the achieved rates may seem low compared to the values seen in modern IP-based networks and forwarding hardware, we have to stress that having an order of 1000 – 2000 Interest packets per second are (1) typical for current ICN solutions, and (2) can result in large transfers of data packets if the replies are instead sent as unicast IP packets that do not have to be rewritten (as is the case in our demonstrator). Furthermore, the switch application on top of which we have built and evaluated our solutions is single threaded, operates fully in the user space and has never been optimized for performance. Hence, we expect higher performance of our proposal in kernel-

Table 3: Experiment 3 – variable packet size. Means and standard deviations calculated over 30 runs. 'TX [P]' stands for 'a number of transmitted packets'

packet size	test set-up	TX (mean) [P]	TX (sd) [P]	test set-up	TX (mean) [P]	TX (sd) [P]	test set-up	TX (mean) [P]	TX (sd) [P]	test set-up	TX (mean) [P]	TX (sd) [P]
78	A	99996	19.0	B	100000	0.0	C	99987	62.6	D	100000	0.0
132	A	100000	0.0	B	100000	0.0	C	99982	62.8	D	99988	63.5
556	A	99974	48.4	B	99996	10.9	C	99977	55.4	D	99993	17.5
1036	A	77221	60.1	B	77209	105.5	C	77201	116.1	D	77226	95.5
1510	A	58534	1404.6	B	59665	758.0	C	59916	1745.8	D	60473	464.9

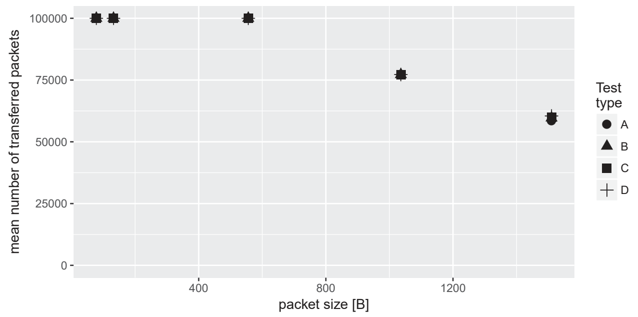

Figure 8: Experiment 3 – variable packet size (no error bars are plotted since they are too small to be visible)

and hardware-optimized switches and conclude that a prohibitive bottleneck introduced through SDN/eBPF is unlikely. In Section 6 we will give our insights on additional performance improvement possibilities.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have discussed how SDN can facilitate a widespread adoption of emerging network technologies such as ICN. To do so, we have exploited the programmability of both control plane and data plane. The critical element to achieve our goals is parsing ICN datagrams by SDN. Such functionality was so far not supported and we proposed and implemented a framework which allows for flexible matching in essentially any protocol, thus being suitable for current and future ICN protocol suites. To achieve this, we have used extensibility features offered by OpenFlow and selected eBPF as a main component of a matching engine. Since eBPF allows to develop matching programs not only in assembly language but also in the (restricted) C language, it greatly simplifies the development process. We have released the source code of our extensions introduced to a switch and controller [29][30], along with a sample ICN-related controller application. We proved the concept by deploying our solution on an intercontinentally distributed testbed, connecting NDN nodes from the USA and the Netherlands over public Internet.

We plan to continue this work by further enhancing functionality (e.g., QoS) and performance with the ultimate goal to implement it as a 'production' component of a testbed like SciNet. Performance improvements can be achieved by, for example, implementation using a switch operating in kernel space. An already available eBPF action patch-set for OVS shows that high performance in-kernel support is possible [9]. Recent developments in the Linux kernel have provided some ways to transparently offload eBPF programs to hardware. The Linux *tc* subsystem has been extended to support this feature in recent patches[2]. This allows directly attaching an (e)BPF program to an interface, which in turn is then offloaded to a specialized NIC [6]. Another initiative the Linux community has introduced is the eXpress Data Path (XDP, [15]) in the kernel, allowing for bare metal packet processing using eBPF. XDP allows hooking into the lowest point in the network stack, before any buffers are assigned, which can provide huge performance boosts depending on the application. XDP does not require any specialized hardware and some vendors have already released specialized drivers that allow for transparent offloading of XDP programs, providing another enhancement possibility, see[16].

An interesting topic is also usage of eBPF not only for matching but for performing the actions as well [1]. In the ICN context, a potential enhancement would be matching using a more general rule, e.g., a regular expression supplied as user parameter. Another direction of investigation is finding which additional features are needed to provide more dynamic programming of the data plane, e.g., provide hooks to (temporary) store data in the switch memory outside of the scope of single packet.

7 ACKNOWLEDGMENTS

We acknowledge the help of former TNO colleagues Borgert van der Kluit and Ray van Brandenburg in the early stage of this research. Also, we appreciate the discussions with Jeffrey Panneman on hardware acceleration possibilities. Furthermore, we would like to thank our submission-shepherd Mayutan Arumaithurai and the anonymous reviewers for their help in improving this paper.

REFERENCES

[1] https://mail.openvswitch.org/pipermail/ovs-dev/2015-February/294542.html.
[2] BPF hardware offload via cls_bpf . https://lwn.net/Articles/689541/.
[3] clang. http://clang.llvm.org.
[4] CPqD OpenFlow 1.3 Software Switch. http://cpqd.github.io/ofsoftswitch13/.
[5] Named Data Networking. http://named-data.net.
[6] Open-NFP. http://open-nfp.org/resources/.
[7] OpenFlow Switch Spec. v.1.3.4. https://www.opennetworking.org/.
[8] P4 language. https://p4.org.
[9] RFC: add openvswitch actions using BPF. http://thread.gmane.org/gmane.network.openvswitch.devel/44109.
[10] Ryu. https://osrg.github.io/ryu.
[11] The CCNx Project. https://www.ccnx.org.
[12] uBPF. https://github.com/rlane/ubpf.
[13] C. Fan, S. Shannigrahi, S. DiBenedetto, C. Olschanowsky, C. Papadopoulos, and H. Newman. Managing scientific data with named data networking. In *Proceedings of the Fifth International Workshop on Network-Aware Data Management*,

page 1. ACM, 2015.

[14] A. K. M. M. Hoque, S. O. Amin, A. Alyyan, B. Zhang, L. Zhang, and L. Wang. NLSR: Named-data Link State Routing Protocol. In *Proceedings of the 3rd ACM SIGCOMM Workshop on Information-centric Networking*, ICN '13, pages 15–20, New York, NY, USA, 2013. ACM.

[15] IO Visor Project. XDP – eXpress Data Path. https://www.iovisor.org/technology/xdp.

[16] Jesper Dangaard Brouer. XDP - eXpress Data Path. http://prototype-kernel.readthedocs.io/en/latest/networking/XDP/implementation/drivers.html.

[17] S. Jouet, R. Cziva, and D. P. Pezaros. Arbitrary packet matching in openflow. In *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6, July 2015.

[18] J. Kicinski and N. Viljoen. eBPF Hardware Offload to SmartNICs: cls bpf and XDP. http://open-nfp.org/documents/1/eBPF_HW_OFFLOAD_HNiMne8.pdf.

[19] D. Kreutz, F. M. V. Ramos, P. E. VerÃŋssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proc. of the IEEE*, 103(1):14–76, Jan 2015.

[20] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference*, USENIX'93, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[21] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comp. Comm. Rev.*, 38, Mar. 2008.

[22] N. B. Melazzi, A. Detti, G. Mazza, G. Morabito, S. Salsano, and L. Veltri. An openflow-based testbed for information centric networking. In *Future Network Mobile Summit, 2012*, pages 1–9, July 2012.

[23] Named Data Networking. NDN Essential Tools. https://github.com/named-data/ndn-tools.

[24] Named Data Networking. NDN Essential Tools. https://github.com/named-data/ndn-tools/tree/master/tools/chunks.

[25] Named Data Networking. repo-ng: Next generation of NDN repository. https://github.com/named-data/repo-ng.

[26] S. Signorello, R. State, J. FranÃğois, and O. Festor. Ndn.p4: Programming information-centric data-planes. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 384–389, June 2016.

[27] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu. Dc.p4: Programming the forwarding plane of a data-center switch. In *Proc. of the 1st ACM SIGCOMM Symposium on SDN*, SOSR '15, pages 2:1–2:8, New York, NY, USA, 2015. ACM.

[28] A. Starovoitov. BPF - in-kernel virtual machine. http://events.linuxfoundation.org/sites/events/files/slides/bpf_collabsummit_2015feb20.pdf, 2005.

[29] TNO. ofsoftswitch13. https://github.com/TNODigitalInnovations/ofsoftswitch13/tree/SDN4ICN. GitHub repository.

[30] TNO. Ryu. https://github.com/TNODigitalInnovations/ryu/tree/SDN4ICN. GitHub repository.

[31] N. L. M. van Adrichem, F. Iqbal, and F. A. Kuipers. Backup rules in Software-Defined Networks. In *Proc. of the IEEE Conference on Network Function Virtualization and Software Defined Networks*, November 2016.

[32] N. L. M. van Adrichem and F. A. Kuipers. NDNFlow: Software-defined Named Data Networking. In *Network Softwarization (NetSoft), 2015 1st IEEE Conf. on*, April 2015.

[33] L. Veltri, G. Morabito, S. Salsano, N. Blefari-Melazzi, and A. Detti. Supporting information-centric functionality in software defined networks. In *2012 IEEE Int. Conf. on Comm.*, pages 6645–6650, June 2012.