




Click-UP: Toward the Software Upgrade of Click-Based Modular Network Function

Junxiao Wang, Heng Qi , *Member, IEEE*, Keqiu Li , *Senior Member, IEEE*, and Steve Uhlig 

Abstract—A Click-based network function (NF) has significant advantages for pipeline development, including modularity, extensibility, and programmability. Despite these features, its internal architecture has unfortunately not kept up with some specific problems of the software upgrade. To motivate our article, we analyzed a series of use cases to identify the limitations of native Click. These limitations include the inefficiencies in modifying modules, integrating modules, and recovering states. To bridge the gap, we present three novel enhancements in our Click upgrade (Click-UP) system: 1) *modular state abstraction* refines each type of stateful operations as an atom operation and decouples it from the pipeline, letting separately managing logics for stateless operations and stateful operations become practical; 2) *essential module integration* manages dependencies between modules, avoiding shipping unnecessary modules with neutral functionalities to the target NF; and 3) *local state migration* migrates needed states seamlessly from the old NF to the target NF at local memory. Our evaluation demonstrates that Click-UP reduces the context code required for module modification by 12–81%, cutting down the NF integration time by 78–96% and the service disruption time by 76–93%, as compared to the software upgrade performance represented by native Click.

Index Terms—Click, modular network function, software upgrade.

I. INTRODUCTION

FOR network function virtualization (NFV), an active network function (NF) typically comes with infrequent but necessary software upgrades, e.g., when off-the-shelf NFs become inappropriate in terms of their functionalities or when NFs evolve [1]. NFs have to be refactored and adapted to fit their new/improved purpose better. With the software-driven pipeline inherent to NFV [2] comes the opportunity to replace traditional (partly) hardware-based upgrade with software one, increasing cost efficiency by allowing technological and software improvements to include faster [3].

Manuscript received October 29, 2019; revised February 10, 2020; accepted February 26, 2020. This work was supported in part by the National Key Research and Development Program of China under Grant 2016YFB1000205, in part by the National Natural Science Foundation of China under Grant 61772112, Grant U1701263, Grant 61672379, and Grant 61751203, and in part by the Science Innovation Foundation of Dalian under Grant 2019J12GX037. (Corresponding author: Heng Qi.)

Junxiao Wang, Heng Qi, and Keqiu Li are with the School of Computer Science and Technology, Dalian University of Technology, Dalian 116024, China (e-mail: wangjunxiao@mail.dlut.edu.cn; hengqi@dlut.edu.cn; keqiu@dlut.edu.cn).

Steve Uhlig is with the School of Electronic Engineering and Computer Science, Queen Mary University of London, London E1 4NS, U.K. (e-mail: steve.uhlig@qmul.ac.uk).

Digital Object Identifier 10.1109/JSYST.2020.2979170

The Click [4], thanks to its design, has been the best platform for the NF upgrade [5]–[7]. Compared to other platforms such as P4 [8], Click encourages modular programming of the packet processing pipeline. Prior work [9] showed that a wide range of NFs share a considerable amount of functionalities. Click makes it easy to reuse such functionalities, abstracting them into a set of reusable modules, called elements. Therefore, the Click-based NF can be refactored by modifications to the required module code rather than to the whole pipeline.

Despite these features, its internal architecture has unfortunately not kept up with some specific problems of the software upgrade. We conclude the limitations¹ represented by Click-based NF upgrade from three perspectives.

L1. Modifying Modules: While the framework (in Click) enables the pipeline in a modular way, the way to identify state is not modular. The developers require modifications to module code to identify needed state, let alone state maintenance, e.g., custom state allocation, track updates to state, and (de)serialize state objects. This process is tedious and manual, hindering the adoption of the stateful upgrade. When the functionalities of the NF are complex, the logic to update/create different pieces of states can be intricate. It will be challenging to make sure the completeness or correctness of manual modifications.

L2. Integrating Modules: Current module integration (in Click) is a time-consuming process. Although the upgrade declared what functionalities it would use for, integration (in Click) does not bound to the required modules, but instead redundantly shipping inessential modules with neutral functionalities to the target NF. The substantial burden involved increases the upgrade latency significantly as well as its further impacts on the service consistency.

L3. Recovering States: Native Click does not provide support for state migration. Due to the absence of needed states at the new instance, the software upgrade (in Click) always leads to incorrect operations conducted by the target NF. For the state migration, there have been two lines of research: 1) checkpointing state regularly into one remote instance and the state migrated from the instance is reconstructed [10]–[12]. State migration from remote, however, takes time, inherently increases the upgrade latency, and ignores the local state update during the checkpointing; and 2) logging all inputs (i.e., packets) and using deterministic replay in order to rebuild the state [13]–[15]. The solution of this kind, in fact, works at the cost of a substantial

¹Note that Click is not designed with an efficient upgrade in mind; hence, these limitations are not aimed at evaluating Click itself, only our Click-based upgrade system.

increase in recovery time (e.g., replaying all packets received since the last checkpoint).

To bridge the gap, we present three novel enhancements in the Click upgrade (i.e., so-called Click-UP) system.

E1. State Operation Abstraction: Click-UP achieves the modular state abstraction with a series of refined atom operations, which are independent of stateless modules and correspond to needed states. Each atom operation implements a single stateful functionality that packets travel through and only requires operators to insert it into the pipeline. For developers, separately managing logics for stateless modules and atom operations become practical and will be relatively simple.

E2. Essential Module Integration: In Click-UP, the target NF is built with only essential modules, which are required by the functionalities of the target NF. By managing the functional dependencies between modules, the module's compilation and linking methods will stick to dependencies, thereby eliminating the burden of redundantly shipping inessential modules with neutral functionalities and significantly reducing upgrade delays.

E3. Local State Migration: Click-UP enables the state migration scheme at the local memory. This scheme relies on a pull/push state interface embedded in each NF to migrate needed states. With a coordinator daemon as an intermediary, internal network states are collected through the pull interface of the old NF and are seamlessly reloaded to the target NF through the push interface. Access to external/shared states is also migrated. In doing so, needed states are migrated effectively in local, significantly eliminating the service disruption.

To demonstrate the efficiency of our solution, we present two typical NFs implemented on top of Click-UP: network address translation (NAT) and whitelist Firewall. Click-UP reduces the context code required for module modification. We also show that NFs on Click-UP conduct the software upgrade with significantly reduced NF integration time and service disruption time, as compared to native Click. Click-UP is open source and is available at <https://click-up.github.io>.

The rest of this article is organized as follows. Section II presents the motivation and design of the modular state abstraction. Followed by Section III, we show how to compile/link the essential modules according to the managed dependencies. Section IV presents the migration of needed states between the old instance and the target instance. In Section V, we evaluate the performance of our upgrade system over a series of testbed simulations and cared metrics analysis. We discuss the current limitations of Click-UP in Section VI. In Section VII, the related work is summarized. Finally, Section VIII concludes this article.

II. MODULAR STATE ABSTRACTION

A. Motivation and Challenges

By rethinking the software upgrade of the Click-based modular NF, we denote its lifecycle with Fig. 1. The lifecycle depicts such a process: developers implement modules with new features and store them at a module library; operators give deterministic upgrade intents, resolving to a list of modules corresponding to required functionalities; and by collecting the

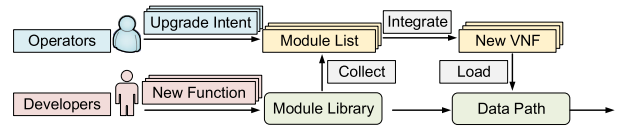


Fig. 1. Lifecycle for the software upgrade of the Click-based modular NF.

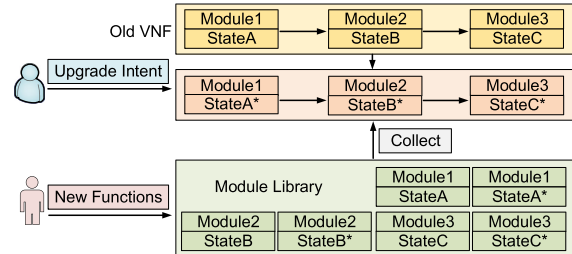


Fig. 2. Traditional software upgrade, where the target NF is organized in a stateless way (i.e., lacking of modular state abstraction).

required modules from the library, target NFs are then formed. On the data path, the old NFs are unloaded and replaced with the target ones.

In domains of Click, the modularity has been developed to allow operators to use high-level abstractions for the software upgrade, while the developers implement those abstractions (ensuring detailed functionalities). The use of the high-level modular abstractions should include the abstractions not only for stateless pieces (e.g., packet header parser), but also for stateful pieces (e.g., connection information lookup). Tight coupling of these pieces easily becomes a source of modification complexity and a maintenance nightmare.

As shown in Fig. 2, traditional upgrades are organized in a manner without modular state abstraction. The target NF is composed of a sequence of directed modules, each of which packages needed states inside. These states are dynamic (they can be updated by each incoming packet) and critical (their values determine correct operation conducted by the NF). Recognizing this, and given the nonmodular way to identify state, developers must modify carefully, or at least annotate, module code to perform custom state allocation, track updates to state, and (de)serialize state objects. These factors make such modifications difficult. When the functionalities of the NF are complex, the logic to update/create different pieces of states can be intricate. It will be challenging to make sure the completeness or correctness of manual modifications. Moreover, these factors make such modifications redundant in the module library. It complicates the dependence management and makes the process of module collecting inefficient.

B. Design and Implementations

To enable the modular state abstraction, Click-UP refines each type of stateful operations as an atom operation and decouples it from the pipeline. As illustrated in Fig. 3, the atom operations correspond to needed states (read or write). Each atom operation implements a single stateful functionality that packets travel through and only requires operators to insert it into the

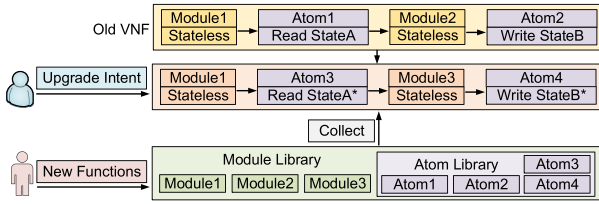


Fig. 3. Software upgrade with Click-UP, where the target NF is organized in a stateful way.

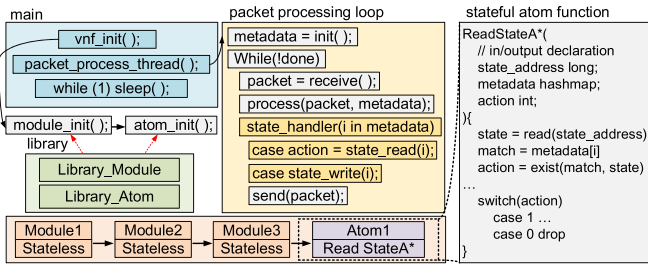


Fig. 4. Processing of an NF using the modular state abstraction.

pipeline and specify its action of output. By decoupling stateless operations and stateful operations, separately managing for them becomes possible. For developers, custom needed states can bypass the module library and direct access to the atom library, which should be relatively simple and with less redundant modifications/management. Note that the meaning of the module is different in the concept of Click and Click-UP. Since Click-UP decouples stateful operations from original Click modules, the module of Click-UP refers to purely stateless operations.

As illustrated in Fig. 4, most of the NF code can be logically divided into three basic parts: initialization, packet receive loop, and packet processing. The initialization code runs when the NF starts. It reads and parses configuration input and loads modules or files. All of these can be done in `main()`. The packet receive loop is responsible for reading a packet from the kernel and passing it to the packet processing procedure. The latter analyzes and potentially modifies the packet. This procedure reads/writes needed states to the processing of the current packet. Via declaring the atom operation in Click-UP, needed states can be identified and embedded to the NF automatically. Each atom operation will come with an application programming interface (API) to read or write state. The focus of the state store is the key-value interface. That is, the API can read values by providing a key (which returns the value) or write values by providing both the key and the value. We leverage this key-value interface to exchange needed states with the data store in a general way, by registering the metadata for each atom operation. The metadata includes all the states carried by each packet that enters pipeline. All the states are also stored using the metadata as the data structure.

Table I shows the states to be decoupled and read/written by the key-value interface for some typical NFs. Each of the NFs contains a certain number of `WriteState` atoms and `ReadState` atoms.

TABLE I
DECOUPLED STATES IN THE ATOM OPERATION FOR TYPICAL NFs

VNFs	State	Key	Value
VPN	Authorised Token	5-Tuple	Token String
	Whitelist	Cluster ID	5-Tuple
Firewall	TCP FSM	5-Tuple	FSM State
	Pool of IPs and Ports	Cluster ID	IP and Port List
NAT	Mapping of Port	5-Tuple	IP and Port
	Automata State	5-Tuple	State Number
IPS	TCP Sequence	5-Tuple	Sequence Number
	Packet Buffer	Buffer ID	Buffered Packet
Load Balancer	Backend List	Cluster ID	Backend IP List
	Assigned Server	5-Tuple	Server IP Address

- 1) *Load Balancer*: Upon receiving a TCP connection request, the atom of `WriteState` retrieves the list of backend servers from data store and then assigns a server to new flow. The load for backend servers is subsequently updated, and the revised list of backend servers is written into the data store. The assigned server for the flow is also stored into the data store before the packet is forwarded to the selected server. For a data packet, the atom of `ReadState` retrieves the assigned server for that flow and forwards the packet to the server.
- 2) *Signature-Based IPS*: Upon receiving a new flow, the atom of `WriteState` initializes the automata state and writes it into data store. The automata state is computed against a database of signatures from known malicious threats. For a data packet, the atom of `ReadState` retrieves the deterministic automaton for that flow. The bytes from the payload are then scanned. In the case of malicious signatures, the subsequent packets are discarded. Otherwise, the packet is forwarded, and the deterministic automaton is updated. Toward the TCP out-of-order problem, the atom of `WriteState` buffers out-of-order packets into the data store. The atom of `ReadState` retrieves buffer to reassemble the stream of bytes.
- 3) *NAT*: Upon receiving a TCP connection request, the atom of `WriteState` retrieves the list of ips and ports from data store and then assigns a pair of (ip, port) to new flow. The load for the pool is subsequently updated, and the revised list of the pool is written into the data store. For a data packet, the atom of `ReadState` retrieves the assigned pair of (ip, port) for that flow and updates the packet header.
- 4) *VPN*: Upon receiving a TCP connection request, the atom of `WriteState` initializes the authorized token for new flow and writes it into the data store. For a data packet, atom of `ReadState` retrieves the authorized token for that flow and decrypts the stream of bytes.
- 5) *Whitelist-Based Firewall*: Upon receiving a new flow, the atom of `WriteState` initializes the TCP finite-state machine (FSM) for that flow and writes it into data store. In the absence of an invalid FSM state, the updated whitelist is stored into the data store for the new flow. For a data packet, the atom of `ReadState` retrieves the whitelist and forwards the packet.

To clarify above mappings, we show how to describe the stateful target NF with atom operations included in the modular pipeline. Take the whitelist firewall as an example (see Fig. 5).


```

1  - - - FromDevice p1          /* hook the packet from vnic p1 */
2  FromDevice - - Classifier 12/0800 /* filter ip packet, check ip type */
3  Classifier 0 - - ToDevice p2 /* non-tcp, directly forward to vnic p2 */
4  Classifier 1 - - Strip 14 /* tcp, strip layer 2 header */
5  Strip 0 - - CheckIPHeader /* IP checksum */
6  CheckIPHeader 0 - - ReadWL /* read whitelist from the data store */
7  ReadWL 0 - - ToDevice p2 /* 5-tuple on whitelist, directly forward to vnic p2 */
8  ReadWL 1 - - ReadFSM /* not on whitelist, read TCP finite state machine */
9  ReadFSM 0 - - WriteFSM /* valid FSM state, write into FSM */
10 WriteFSM 0 - - ToDevice p2 /* update FSM and directly forward to vnic p2 */
11 ReadFSM 1 - - Discard /* invalid FSM state, drop the packet */
12 ReadFSM 2 - - WriteWL /* FSM state is established, write into whitelist */
13 WriteWL 0 - - ToDevice p2 /* forward to vnic p2 */
14 - - - FromDevice p2          /* hook the packet from vnic p2 */
15 FromDevice - - ToDevice p1 /* directly forward to vnic p1 */

```

Fig. 5. Stateful target NF of a whitelist-based firewall. The text corresponds to a DAG using stateless modules and stateful atom operations as nodes.

It employs `ReadFSM`, `ReadWL`, `WriteFSM`, and `WriteWL` as stateful atoms. When a packet arrives at the input port, it is processed by `FromDevice`, `Classifier`, `Strip`, and `CheckIPHeader` in order. These modules belong to the stateless part of the pipeline, responsible for header field checking and filtering. After that, `ReadWL` retrieves the whitelist for the packet. If its 5-tuple is on the whitelist, the packet is forwarded to the output port. Otherwise, the direct forwarding is declined, and `ReadFSM` retrieves the transmission control protocol (TCP) FSM for the packet. If its TCP tag is invalid, the packet is dropped. Otherwise, `WriteFSM` updates new FSM into the data store. If its TCP tag is valid and the connection is established, `WriteWL` updates the whitelist for that flow, before the packet is forwarded to the output port.

Besides the example, we argue that the modular state abstractions in Click-UP are general for more NFs and have the efficiency of their implementation and maintenance. We allow the developers to customize the stateless modules and the stateful atoms they need. This gives the NF upgrade great flexibility while allowing the operators to use optimized implementations of these abstractions. By refining state operations into modular atoms, the integration and the state migration will be more beneficial and effective. In Section III, we show how to integrate the modules and atoms determined by the target NF. In Section IV, we introduce how to migrate needed states (related to the atoms) from the old NF to the deployed new NF.

III. ESSENTIAL MODULE INTEGRATION

A. Motivation and Challenges

For operators, it is ideal to achieve always up-to-date NFs and zero upgrade latency. For maximum security, a cellular provider may want traffic always to be processed by the latest NF. For example, an service level agreement (SLA) may require that outdated NF instances never process traffic for more than 10 min per year [16]. A shorter upgrade delay let operators' intents be applied to data plane sooner.

Click's native module integration process can be divided into two steps: 1) compile and link all modules available in the module library into an executable NF; and 2) load needed modules of the executable NF to the pipeline. However, for the software upgrade, required new functionalities are always outside the range of the executable NF. As a result, each upgrade

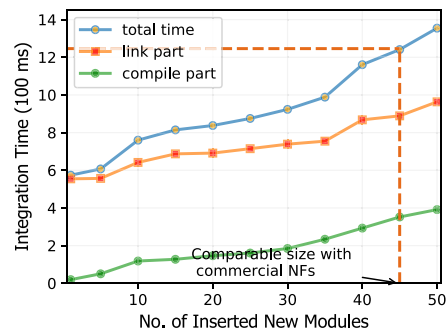


Fig. 6. Integration time of the software upgrade in Click. The data are obtained in a server with Intel Xeon (E5-2630v3) 64-GB RAM.

will lead to integration between the new modules and all other modules. So what impact does this NF integration have on the upgrade latency, even the service consistency?

To answer the question, we used a Click-based NAT implemented by Mazu Networks and upgraded it with a given number of customized `Print` modules² inserted into the original pipeline. The integration time of the target NAT was measured by observing the timestamp of the first printed flow. In Fig. 6, we plotted the average of 20 experiments.

We find that the time for module compilation and linking covers up to 99% of the total integration time. We observe an upward trend of compilation time as the number of new modules increases, while the time for linking new modules has a less pronounced upward trend, more constant. The reason behind is that Click natively lets the executable NF contain the whole networking stack. Hence, all existing modules (up to 300+) are linked every time redundantly, even if we add/modify a single module to the pipeline. The burden makes the upgrade latency to be significantly increased. Toward more strong service consistency, we are motivated to care about the functional dependencies between the modules and let every integration stick to the required functionalities.

B. Design and Implementations

To cut down unnecessary burden involved by module integration, we propose a more lightweight integration in Click-UP.

²The print module does nothing but log the timestamp of packet traversing through it. Each print module is implemented with a Click element.

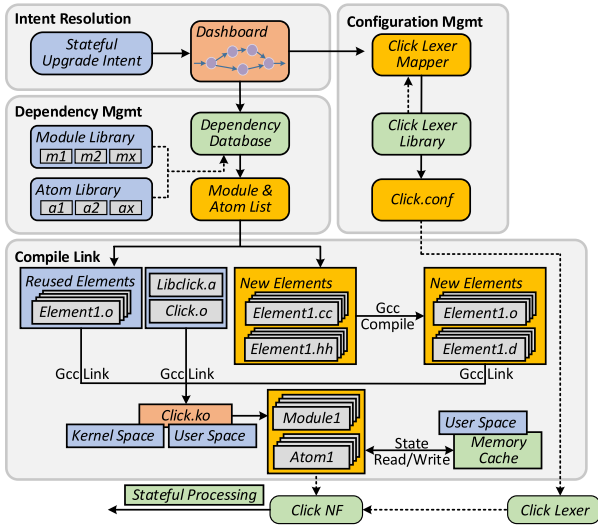


Fig. 7. Design of lightweight integration layer in Click-UP.

Keeping in mind what the target NF will be used for, integration in Click-UP sticks to the required modules, instead of redundantly shipping inessential modules with neutral functionalities to the target NF. To this end, we present the design and implementation of an integration layer on top of Click and modify the way modules compiled and linked. As shown in Fig. 7, the components of the integration layer are *intent resolution component* (IRC), *dependence management component* (DMC), *configuration management component* (CMC), and *compile link component* (CLC).

1) *Intent Resolution Component*: IRC is with an upgrade intent resolver. The resolver is input with a stateful upgrade intent (as illustrated in Fig. 5) and then resolve it into directed acyclic graphs (DAGs) of the target NF, where the DAG corresponds to a stateful packet processing pipeline.

2) *Configuration Management Component*: CMC is with a Click lexer mapper, which maps the pipeline of the target NF into configuration files `Click.conf` of Click lexer.

3) *Dependence Management Component*: DMC is with a dependence database, which stores the module relationship used by dependence exploration. In most cases, the declared modules in the target NF are not equal to the required modules in the executable NF. The traditional scheme in Click directly embraces all the modules into the executable NF, which has been proven to be inefficient. To manage the functional dependencies between the modules and let integration stick to the required functionalities, we leverage a functionality-oriented element dependence model as follows.

We have two classes of dependencies defined in the modularity of Click: element³-to-functionality and functionality-to-functionality. When an element depends on a functionality without itself providing functionalities, we refer to the dependence as an element-to-functionality dependence. When the element

³“element” is a term in the concept of Click, indicating a modular component in the pipeline. In the concept of Click-UP, the terms “module” and “atom” are to indicate a purely stateless Click “element” and a purely stateful Click “element,” respectively.

provides functionalities and requires other functionalities to provide its own functionality, we refer to the dependence as a functionality-to-functionality dependence.

The required functionalities of an element instance are bound to dependencies once this instance is employed. Any functionalities provided by this instance are registered after the dependencies of this instance are satisfied. An element instance is valid when its functionality dependencies are fully satisfied. Following this, an element instance is always in one of two possible stages: invalid or valid. The invalid stage means that at least one of its functional dependencies is not satisfied. The end of dependence exploration is with all the declared element instances of the target NF in a valid stage.

4) *Compile Link Component*: CLC is with a redesigned element compiler and linker. Note that we implement both the stateless module and the stateful atom with unified Click modular files `Element.cc` and `Element.hh`. Therefore, input of the CLC (i.e., output of the DMC) is a set of Click elements. We first categorize these elements into two boxes: 1) elements used by both the old NF and the target NF; and 2) elements only used by the target NF. For the elements in box 1, our compiler will reuse their compiled files `Element.o` to reduce overhead, while for the elements in box 2, our compiler will normally compile their code files `Element.cc` and `Element.hh` into `Element.o` plus `Element.d`. In the step of linking, our linker will link three parts of files into an executable NF file `Click.ko`: a) a set of reused elements; b) a set of fresh elements; and c) a set of static resource files, e.g., `Libclick.a` and `Click.o`.

According to environmental requirements, the NF file `Click.ko` has two deployable versions, which run in Linux kernel space and user space, respectively. In both of two cases, the atoms in the target NF will read/write the needed states (via the stateful operations defined in Section II) within a memory cache of user space. In Section IV, we show how the scheme of state recovering works for Click-UP.

IV. LOCAL STATE MIGRATION

A. Motivation and Challenges

Besides the period of service inconsistency, another metric cared by operators is the period of service disruption. When the target NF is deployed, whether it can immediately handle traffic and does not disrupt the data plane is essential. An upgrade without state management always leads to incorrect operations conducted by the new NF, due to the absence of state at the target instance. This may involve states such as connection information in a stateful firewall, substring matches in an intrusion detection system, address mappings in a NAT, or server mappings in a stateful load balancer.

Click modularity poses significant challenges in recovering network states during the software upgrade. To better quantify the impact of state loss on the data plane, let us consider files of 0.5, 1.0, and 2.0 GB transmitted by an http server to a client through the previous NAT. We upgraded the NAT with a customized `Print` module during the file transmission. As shown in Fig. 8, the transmission duration (*y*-axis) is based on the average across 20 experiments.

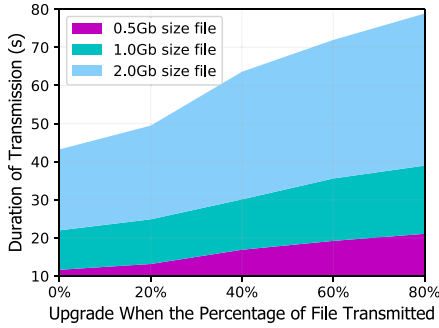


Fig. 8. Transmission time with the state loss. The data are obtained in an http server with 1.0-Gb/s NIC.

We find that the ACK packet sent by the client had its source port randomly overwritten after the port mapping tables were lost by the target NAT so that the rewritten port was inconsistent with the one stored in the old NF. This caused the wrong source port number to be used, so the server reset the connection. If the connection is reset due to the upgrade, the original transmission progress will also be reset to the beginning of the file, wasting the previously transmitted subset of the file. When no upgrade happened during the file transmission (x -axis coordinate is 0%), the average total transmission time was 11 s for the 0.5-GB size file and 43 s for the 2.0-GB file. When the upgrade happened after 40% of the file has been transmitted, the total average transmission time became 16 and 63 s, respectively. Obviously, due to the state loss, the larger the file, the more unnecessary traffic retransmission, potentially causing more impact on the data plane, such as switch buffer pressure, packet loss, and congestion. In order to avoid service disruption, we are motivated to take care of Click’s state migration during the software upgrade.

B. Design and Implementations

The main idea of state recovering is to reserve the states of the old NF and seamlessly migrate it to the target NF, thus eliminating service disruption. We present the following requirements that an ideal migration scheme should satisfy.

- 1) *Low Performance Overhead*: The NF is often on the critical data path, processing millions of packets per second. The performance overhead involved by the state maintaining scheme (latency and throughput) on individual flows must be minimal.
- 2) *Low Migration Latency*: The state migration as one stage of software upgrade, whose duration must be minimal. Much too long time used for migrating states may top up a significant upgrade latency and its further impacts on service consistency.
- 3) *Recovery Transparency*: The NF is often an invisible entity that lies along the network path between two endpoints. Thus, it is insufficient to just steer flows to the target NF transparently. The states of the old NF must be recovered with consistency, such that on-flying flows can continue with minimal disruption.

Click-UP employs a new approach to state migration. Instead of checkpointing and migrating states from remote, we capitalize

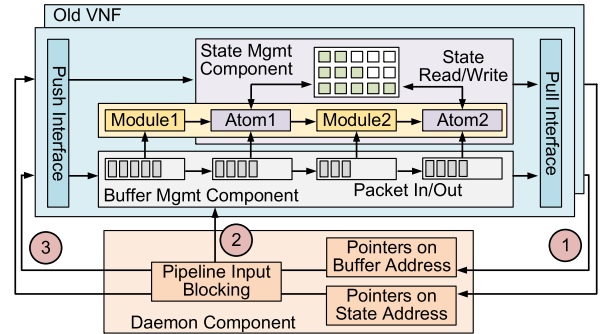


Fig. 9. High-level design of state migration in Click-UP.

on a unique structure of the NF to enable state migration at the local memory. Fig. 9 shows the high-level components that make up the migration scheme. We assume that the target NF and the old NF are on the same machine. We also assume that needed states include internal states and external/shared states (e.g., clock signal), where the latter only requires to migrate the access interface.

The NF runs three components for migrating states.

1) *State Management Component (SMC)*: SMC is for managing atoms’ stateful operations and controlling access to a set of key–value states that are being processed at the memory cache. It also plays a role of responding calls from pull/push interface, fetching or embedding needed states during upgrade.

2) *Buffer Management Component (BMC)*: BMC is for ensuring that packets enter and exit the NF at appropriate times. The incoming and outgoing packets through the NF will be buffered in a blocking queue tree, where unless all the leaf queues go empty, the root queue remains blocked. This scheme can avoid inconsistencies on the buffer as well as the states, since the packets buffered in leaf queues are typically abandoned in the target NF. The balance here, for BMC, is that if the queue size is set too large, the process of blocking may induce higher latency because more packets need to be drained from the leaf queues, and if the queue size is set too small, the packets may lose the opportunity for buffer gains. In Click-UP, we have a fixed queue size, but we ultimately envision an BMC, which increases or decreases the queue size adaptively. The BMC can respond calls from pull/push interface, fetching or embedding packet buffers during upgrade.

3) *Coordinator Daemon Component*: It is for coordinating the workflow of state migration during upgrade. Upon informed by an event that the upgrade set up, coordinator daemon component (CDC) will asynchronously call the SMC and the BMC via the pull interface to fetch buffer and states from the old NF (see ① in Fig. 9). Upon informed by an event that target NF has been built finished, CDC will first call the BMC to block the old NF (see ② in Fig. 9), then via the push interface to embed buffer and states into target NF (see ③ in Fig. 9), and, finally, data path will be switched from the old NF to the new one, and the traffic will also be redirected to the target NF.

To optimize this scheme to match the common structure of network processing, we make use of the following three techniques.

1) *Eliminating Data Copy*: To guarantee efficiency and consistency of migration, we do not conduct data copy for any buffer or states, but instead provide a pointer to the memory data store to which issued the pull request.

2) *Memory Pool*: When submitting state and buffer read/write requests to the data store, memory must be allocated for the request. As this process is in the critical path, we, thus, reduce the overhead for allocating memory by having a preallocated pool reused in the memory data store.

3) *Rollback*: Click-UP does compose, compile, and link target click.ko offline (i.e., without disrupting services during this period), and then unload the old click.ko and reload the new one. To roll back from failures (e.g., the target NF startup failed), we employ a status checking, which checks the bootstrap status of the target NF before embedding. If the checking is passed, normally embed buffer and states. Otherwise, it turns to rollback, and the data path will not be switched, avoiding service disruption caused by integration failures.

The SMC and all the states are resided in Linux user space for security and scalability. Remember that Click-UP supports the pipeline running both in kernel and user spaces. In the case that the pipeline is running in kernel space, we leverage the Netlink socket for interprocess communication both between the kernel and user space processes, and between different user space processes. In the case that the pipeline is running in user space, we leverage Netmap to bypass the protocol stack in Linux kernel, which capacities the zero-copy technique.

The target NF can be deployed and hosted with a variety of approaches, such as virtual machine, container, or even a physical machine. We focus on the container as our main deployable unit (matched with pipeline in user space mode). This is due to its fast deployment, low-performance overhead, and high reusability. The old and target NFs are implemented as Docker instances with independent cores and memory space/region. In doing so, we ensure that the switching pipeline does not affect each other. For network connectivity, we share the physical interface among each of the containers (pipelines). Toward this, we use OpenvSwitch to provide virtual interfaces to each container and steer the flows of traffic to the correct NF by installing the appropriate OpenFlow rules.

V. EVALUATION

In this section, a series of testbed simulations were conducted to evaluate the performance of our upgrade system. We seek to answer some questions as follows: 1) What are the advantages of lightweight integration? 2) What is the performance of the local state migration? and 3) How does the impact of the software upgrade system on the data plane? In order to ensure the generality of our upgrade system, we reimplement two typical NFs on top of Click-UP and show that their upgrades perform well compared to their native versions.

A. Simulation Setup

1) *NFs*: To stress the generality of our modular state abstractions, we implemented two typical NFs from the range of native Click.

TABLE II
NUMBER OF MODULES AND ATOMS IN THE EXPERIMENTAL NFs

Network Functions	Click		Click-UP	
	M	A	M	A
Mazu NAT	20	0	20	4
Whitelist Firewall	28	0	28	4

TABLE III
SOFTWARE UPGRADES IN THE SIMULATIONS

Software Upgrades	Updated Operations	Lines of Context Code
Click NAT1	extract origin ip, port (stateless operation)	168
Click-UP NAT1		146 (-13%)
Click NAT2	push mapped ip, port (stateful operation)	193
Click-UP NAT2		36 (-81%)
Click Firewall1	extract 5-tuple, tcp flag (stateless operation)	207
Click-UP Firewall1		183 (-12%)
Click Firewall2	discard invalid packet (stateful operation)	232
Click-UP Firewall2		49 (-79%)

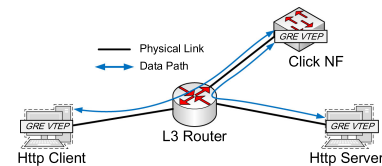


Fig. 10. Topology of the simulations. The data path is implemented with the GRE technique.

- NAT*: It is based on Click Mazu NAT, a modular network address translation implemented by Mazu Networks, and commonly used in academic research.
- Firewall*: It is based on whitelist firewall, implemented in Click; the firewall performs a linear scan of a TCP connection whitelist to find the first matching entry.

Table II shows the number of essential modules in the experimental NFs. As illustrated in Section II, in the implementation of Click-UP, stateful operations of the NF have been decoupled from the pipeline into the refined atoms.

2) *Software Upgrades*: We conducted the simulations using four types of software upgrades. Each NF corresponds to two types of upgrades that update either stateless operations or stateful operations. Table III shows the lines of context code to correspondent upgrades. We find that the modular state abstractions of Click-UP significantly reduce the amount of code involved by updated operations, especially when the updates happen on stateful operations. This proves its benefits on completeness and correctness of manual modifications to the code. We offline finished the coding of updated operations, before all the upgrades happen.

3) *Topology*: We evaluated the performance of software upgrade using a topology shown in Fig. 10. The node of http client will request the node of http server for downloading files. The file stream traverses the Click NF and gets the stateful processing. To enable the data path in this figure, we leverage generic routing encapsulation (GRE) tunnels by creating virtual tunnel end points (VTEPs) on the nodes of http client, http server, and Click NF. All the upgrades happen during the procedure of the file streams.

4) *Devices*: We consider two dominating scenarios to deploy NFs: telecom network and edge network, whose features are

TABLE IV
DEVICES IN THE SIMULATIONS

Simulated Deployment Scenarios		Device Hardware
Telecom Network (TN)	CPU	Intel Xeon E5-2630v3
	RAM	DDR4 64GB
	Router	Pica8 P-3297 1G
Edge Network (EN)	CPU	Broadcom BCM2837
	RAM	DDR2 1GB
	Router	TL-WDR5620 300M

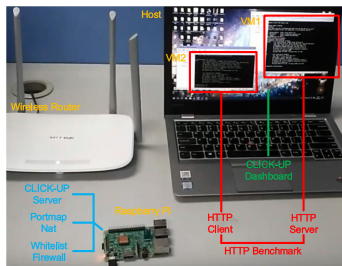


Fig. 11. Deployment environment in the simulated scenario of the edge network.

distinct. The devices in the former case always equip with high-performance hardware, aiming at high network throughput. However, the devices in edge networks are prone to focusing on large-scale deployment (more close to users) and deployment costs; these devices tend to equip with economic hardware. We simulate to deploy Click-UP in two scenarios. Table IV shows the correspondent device hardware. The nodes of http client and http server are deployed on two virtual machines. Fig. 11 shows our testbed environment in the simulated scenario of edge network. Find the demo video at <https://www.youtube.com/watch?v=5G244I0LEYg>.

5) *Schemes Compared*: In terms of module integration, we selected integration scheme of native Click as the baseline. Compared to Click-UP, the scheme in native Click will integrate all modules into an executable NF.

In terms of state migration, we selected two typical schemes from the literature: 1) *remote synchronization* [10]–[12]. When the upgrade happens, the old NF first sends its states to a remote instance, followed by the target NF getting back needed states from the same remote instance; and 2) *packet replay* [13]–[15]. When the upgrade happens, the old NF first sends its states to a remote instance and, then, let the packets arriving later be logged; after the target NF gets back needed states from the same remote instance, the logged packets are preferentially replayed to the target NF (before the target NF start processing packets from the buffer).

B. Metrics Cared

1) *Integration Time*: It is the time for finishing the building job of the target NF. After only the integration is finished, the target NF can then reload the network states. Thus, this metric can reflect an upgrade system's ability to transition an old NF to a new one. Less integration time, more beneficial for always achieving up-to-date NFs, and upgrade intents can be applied to data plane sooner.

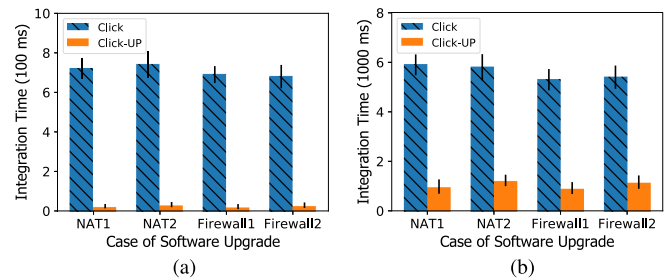


Fig. 12. Comparison of integration time between Click-UP and native Click in two simulated scenarios. (a) TN. (b) EN.

2) *Service Disruption Time*: It is the time topped up by state inconsistencies. With the inconsistent states, the target NF will conduct incorrect operations after going online, resulting in end-to-end service performance degradation. Less service disruption time, more transparent the state recovery is, and the negative impact on the data plane service can be slighter.

C. Effectiveness of Module Integration

To analyze the impact of our essential module integration on the integration time, we conducted an experiment using the predefined upgrade cases: NAT1, NAT2, Firewall1, and Firewall2. For comparison, we started the integration job of the same upgrade case at the same time for native Click and Click-UP and measured their time to complete the job. To emphasize fairness, we let the state migration also included in the integration job of Click-UP.

With the measured integration time, Fig. 12(a) and (b) plots a comparison between Click-UP and native Click under two simulated scenarios, where the case of upgrade varies in the x -axis. We made the following observations.

- 1) Compared with the native version, Click-UP has a significantly reduced integration time. This is expected since Click-UP leverages the dependence management to exclude the redundant modules from the target NF. Therefore, no matter for which simulated scenario, the number of linked modules is significantly reduced.
- 2) Compared with the simulated scenario EN, TN has significantly less integration time (26 ms versus 941 ms) and higher reduction rate (96.1% versus 78.2%). The reason is that powerful computation performance (from, e.g., CPU and RAM) speeds up the process of compiling and linking, thus boosting the gap between Click-UP and native Click.

D. Effectiveness of State Migration

We also investigated how our upgrade system performs in terms of state migration. As we discussed in Section IV, our target instance can seamlessly handle the redirected traffic from the old instance without causing any disruption for the traffic. To illustrate this effect and compare to the traditional approach, we performed a number of file downloads that go through a firewall and a NAT, respectively, and measured the number of successful file downloads and the time required to complete all of the downloads in the following two cases: the firewall and the



Fig. 13. Comparison of time taken to satisfy completed requests between Click-UP and native Click. (a) Click NAT1 and Click-UP NAT1. (b) Click Firewall1 and Click-UP firewall1.

NAT on top of native Click and Click-UP 1) without software upgrade and 2) with software upgrade, where we redirect traffic from the old NF to the target one. For fairness, we simulated all the streams start at some same time, and all the upgrades also start at some same time.

We conduct the experiment using the simulated scenario TN. Fig. 13(a) and (b) shows our results where we download up to 300 10-MB files in a loop of 60 concurrent http downloads through the firewall and the NAT. Fig. 13(a) provides a comparison of time taken to satisfy completed requests through an upgraded NAT. As we can see, much more service disruption time is caused during the software upgrade with native Click. The associated ACK packets sent by the client get their source port randomly overwritten after the port mapping tables are lost on the NAT so that the rewritten port is inconsistent with the one stored in the old instance. This causes the wrong source port number to be used, so the server resets the connection. As a result, breaking existing connections will reset the file transfer, and all the progress before that gets waste. In contrast, there is almost no service disruption time for Click-UP.

Similarly, as plotted in Fig. 13(b), the firewall of native Click is significantly affected by the software upgrade because the new instance does not recognize the redirected traffic, hence drops the connections, which, in turn, results in the client reinitiating the connections after a TCP connection timeout. We find that the Click-UP firewall benefits from seamless state migration, and its corresponding download time is almost the same as the download time without a software upgrade.

E. Insight of Local State Migration

It is critical to understand the effect of the state migration scheme as it may top up the upgrade latency and disrupt the data plane service. We claim that the local state migration of Click-UP provides seamlessness, where much less migration time is achieved with almost no disruption to the traffic. To evaluate the capability of seamless state migration, we performed the following experiment: we stream continuous traffic of http flows through a firewall, keep the flow rate steady, and simulate an upgrade to start at some specific time.

The migration time and the disruption time heavily depend on the flow rate because the flow rate has a significant impact on the number of stored states. We experimented using the simulated scenario TN. Fig. 14(a) and (b) plots a comparison between

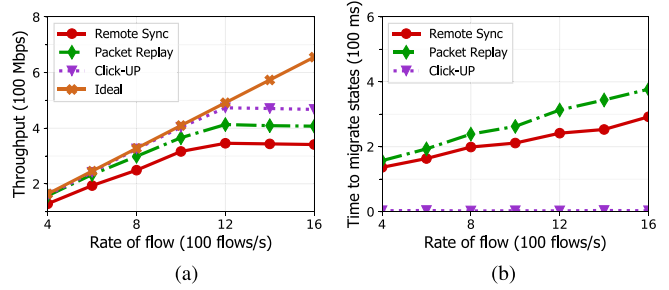


Fig. 14. Comparison of throughput and migration between our local migration schemes and other baseline schemes. (a) Comparison of throughput. (b) Comparison of migration time.

our local migration scheme and other baseline schemes, where the flow rate varies in the x -axis. Each flow consists of 100 packets, and the size of each packet is 512 bytes. Fig. 14(a) provides the measured throughput during the software upgrade. The four curves in this figure represent the ideal throughput (Ideal) that matches the flow rate, migrating states by the remote synchronization (Remote Sync) scheme, migrating states using the remote synchronization scheme followed by packet replay (Packet Reply), and migrating states using the local state migration (Click-UP) scheme.

We made the following observations.

- 1) The throughput of Click-UP matches the ideal one when the rate of flow is lower than 1100 flows per second. It means that the newly added firewall instance normally has the states to process the redirected packets and, therefore, does not get affected by traffic redirection. As the rate of flow increases beyond 1100 flows per second, the effective throughput achieves a bottleneck. This is caused by some time-consuming operations, e.g., IP header checksum in the firewall pipeline.
- 2) The throughput of remote synchronization is, on average, lower than other schemes. The reason is that the states acquired from the remote are inconsistent with actual states. During the remote state transfer, the local states are still being updated in the old NF (the traffic is not yet redirected to the target NF). Upon the traffic is redirected, the new instance starts dropping packets because it does not recognize them (i.e., does not have states for those flows) and, thus, breaks the connections. As the rate of flow increases, the number of associated states will increase accordingly, and the gap in the throughput will become more significant.
- 3) The throughput of Click-UP always outperforms the packet replay, and the gap increases with the flow rate. This is because logging and replaying packets lead to extra computation occupation, while the total computation capacity is fixed. Therefore, in the period of packet logging/replaying, the throughput of the old instance is significantly reduced. As the rate of flow increases, the number of logged/replayed packets will increase accordingly, and the effect on reducing throughput will be more potent.

Fig. 14(b) plots a comparison of time to migrate states between Click-UP and other schemes. Observe that Click-UP

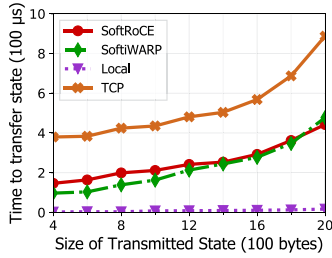


Fig. 15. Comparison of state transmission delay.

always outperforms other schemes, and the migration time of Click-UP has no significant change with increasing flow rate. This is due to the local pointer copy instead of the remote data copy, whose time consumption is independent of the rate of flow and the number of states. Compared to Click-UP, remote synchronization and packet replay both depend on costly remote data copy; their migration time is sensitive to the number of transmitted states as well as the transmission delay. The number of transmitted states is positively correlated with the flow rate.

As for the transmission delay, we implemented two baseline for evaluation. The first one is to transfer state leveraging standard TCP, and the second one is based on software-driven remote direct memory access (RDMA) [17], i.e., SoftiWARP [18] and SoftRoCE [19]. Fig. 15 plots a comparison of state transmission delay between the baseline and our local solution, where the state size varies in the x -axis. We measured the average over 20 tries. The results show that, even though the transmission delay can be largely masked by applying some advanced transmission techniques, there is still a gap compared to the performance in local.

Back to Fig. 14(b), we also observe that the migration time of packet replay is several milliseconds higher than remote synchronization, and the gap increases with the flow rate. This is due to the increasing number of replayed packets. Combined with the results in Fig. 14(a), packet replay has a higher migration time and a higher throughput than remote synchronization. To sum up, Click-UP using the local state migration has significant advantages in terms of throughput and the time to migrate states.

F. Analysis of Data Plane Latency

The interaction with the local state database can increase the latency of each packet, as every incoming packet needs to be blocked until its stateful operations are completed. To evaluate the delay increase, we compared the round-trip time (RTT) of each packet in Click-UP, native Click, and P4-bmv2. We performed the following experiment: the traffic of http stream starts from the http client, travels through a firewall, reaches the server, and is sent back to the client. The client records the sending time and the receiving time of every packet to compute the packet RTT. Fig. 16(a) and (b) shows the RTT of ten flows traversing the firewall under two simulated scenarios. We measured the average over 20 tries.

Fig. 16(a) plots a comparison under the simulated scenario TN. Observe that the extra RTT caused by Click-UP is less than P4-bmv2 (3.2% versus 7.7% on average). Fig. 16(b) plots

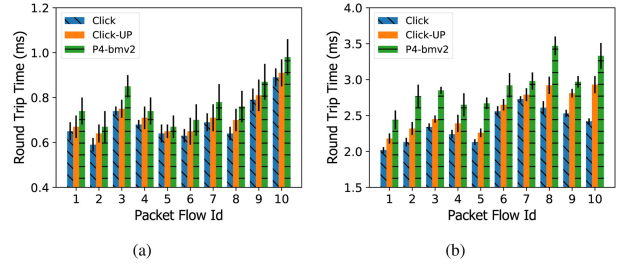


Fig. 16. Comparison of data plane latency between Click-UP, native Click, and P4-bmv2 in two simulated scenarios. (a) TN. (b) EN.

a comparison under the simulated scenario EN, and a similar observation can be found under EN (8.9% versus 11.5% on average). We further investigated that the added latency of Click-UP in the former case is 88.7% less than the latter case on average. This is due to the more luxurious computation performance used in the former case. Overall, the data plane latency caused by Click-UP is bounded by the practical hardware, while the increase rate is strictly limited as compared to the native version.

VI. DISCUSSION ON LIMITATIONS

1) *Distributed Deployment*: In this article, we assume that the target NF and the old NF are on the same machine. We also assume that the internal states are all locked into the single machine. All the typical NF's software upgrades we know of fit comfortably within the assumption range. However, one can imagine the target NF that would be deployed to another machine, or the internal states are distributed. We believe that there may be further opportunities to optimize Click-UP for this case through logically centralized synchronizing and distributively shared states. While our focus in this article is more on single-instance design, and all-in-one solution, as a future direction, we intend to further understand how Click-UP can be adapted to suit the needs of the distributed deployment.

2) *Module Redistribution*: Click-UP decouples stateful operations from Click, allowing developers to customize stateless modules and stateful atoms they need separately. However, for legacy Click modules that contain stateful operations, the developers need to redistribute them as stateless versions, plus correspondent state atoms. The redistribution of a Click module follows three steps: first, find the stateful operation from the pipeline and replace it with correspondent Click-UP atom (as illustrated in Section II); second, remove the stateful operation from the pipeline and build the remaining stateless operation as a Click module; and third, let a mapping relationship set up between them. The module redistribution may, in fact, take an amount of time. Even so, the state operation abstraction and stateful upgrade can help developers and operators capitalize efficiency further on service modification and maintenance, which are, at the end, important for the software upgrade.

3) *Element Splitting*: Currently, in Click-UP, each atom is implemented with an individual Click element. This lets a redistributed NF contain more essential elements, compared to that before. This is because an original Click element (with state operations inside) might be redistributed and split into one purely

stateless element (i.e., module) plus one purely stateful element (i.e., atom). Theoretically, this element splitting poses a certain negative impact on upgrade latency. However, since the number of essential elements and the number of inessential elements are not in the same order of magnitude, the above impact can be ignored. We further demonstrate Click-UP's integration performance in Section V.

VII. RELATED WORK

In this section, we categorize the existing work most related to Click-UP into the following parts: 1) click modular integration; 2) state migration scheme; and 3) other click around enhancement.

1) *Click Modular Integration*: Click natively depends on a modular integration scheme named hotconfig [20]. It integrates all modules to build an executable NF file and write a Click-language description to this file to hot-swap between required network functionalities. During the process of hot-swap, the packets queued in the old NF can be migrated into the new one. However, this scheme still fails to suit the cases of the software upgrade due to the following reasons. First, hotconfig cannot hot-swap required network functionalities out of the scope of integrated modules. As a result, upgrades will inevitably lead to a reintegration involving all existing modules and the inserted/modified new modules. Second, hotconfig, although shifts packets in queues from the old NF to the new one, does not deal with internal network states. In contrast, Click-UP employs dependence management to considerably cut down the overhead of reintegration and provides a seamless state migration.

2) *State Migration*: Existing work about NF state migration mainly focuses on how to deal with the case of failover rather than the software upgrade. Due to different focus points, one can think of the relationship between them to be analogous to the difference between a planned event and an unexpected event. When people consider the failover, they need to assume that the old NF is failed and only can rely on the checkpoints from the remote monitor, or the logged traces in the storage. Accordingly, Pico [10], Split/Merge [11], and OpenNF [12] belong to the first type of work that rely on migrating the states from remote. FTMB [13] and REINFORCE [15] turn to pick another way by replaying traces to migrate states. However, these solutions, although work well in the case of failover, are not best suitable for the software upgrade because they are independent of module integration. Owing to the planned nature of the software upgrade, Click-UP employs a local state migration, which is fused to the integration process and fit a more seamless effect. Note that Click is not designed with the failover or even scaling-out, but instead plays a complementary role in combination with those dedicated solutions.

3) *Click Around Enhancement*: There are also a series of enhanced solution conducted around native Click. We categorize them into two main threads. The first thread belongs to performance-oriented enhancement: ClickOS [2] presents a 5-MB, fast boot, high-performance, Click-driven virtualized software middlebox platform on a commodity server. The fastClick [21] system exploits netmap and dpdk to leverage

the power of hardware multiqueues, multicore processors, and nonuniform memory access on a commodity server. The other thread belongs to function-oriented enhancement: CliMB [22] adds the features of TCP support and blocking I/O into the original Click. Legofi [23] designs and implements a Click-driven functional decomposition for WiFi. Augustus [24] implements a software architecture for ICN routers on top of Click. To our best knowledge, Click-UP is the first enhanced solution tailored to software upgrade.

VIII. CONCLUSION

In this article, we investigated the limitations of the Click internal architecture when facing with the software upgrade. These limitations include the inefficiencies in modifying modules, integrating modules, and recovering states. Motivated by the problem, we presented the design and implementation of Click-UP, a Click-based software upgrade system for the modular NFs. Based on native Click, we made three main improvements in Click-UP.

First, we achieved the state abstraction with a series of stateful atom operations, which are independent with stateless modules and correspond to needed states. For developers, modifications/management to module code to identify needed state, state maintenance, e.g., custom state allocation, track updates to state, and (de)serialize state objects become practical and will be relatively simple. Our evaluation also demonstrated that Click-UP reduces the context code required for module modification. Second, we achieved the essential module integration with the management of functional dependencies. The burden involved in redundantly shipping inessential modules with neutral functionalities is eliminated. The experiment results showed that the integration time is significantly cut down. Third, we achieved the state migration scheme at local memory. The internal states are collected from the old NF and reloaded into the new NF with seamlessness. The experiment results showed that there is almost no service disruption time with Click-UP.

REFERENCES

- [1] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Commun. Surv. Tut.*, vol. 18, no. 1, pp. 236–262, Jan.–Mar. 2016.
- [2] J. Martins *et al.*, "Clickos and the art of network function virtualization," in *Proc. USENIX Symp. Netw. Syst. Des. Implement.*, 2014, pp. 459–473.
- [3] G. Sun, G. Zhu, D. Liao, H. Yu, X. Du, and M. Guizani, "Cost-efficient service function chain orchestration for low-latency applications in NFV networks," *IEEE Syst. J.*, vol. 13, no. 4, pp. 3877–3888, Dec. 2019.
- [4] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.
- [5] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming slick network functions," in *Proc. ACM SIGCOMM Symp. Softw. Defined Netw. Res.*, 2015, pp. 1–13.
- [6] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A software-defined framework for developing, deploying, and managing network functions," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 511–524.
- [7] B. Li *et al.*, "ClickNP: Highly flexible and high performance network processing with reconfigurable hardware," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 1–14.
- [8] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

- [9] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2012, pp. 323–336.
- [10] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Proc. Annu. Symp. Cloud Comput.*, 2013, pp. 1–15.
- [11] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2013, pp. 227–240.
- [12] A. Gember-Jacobson *et al.*, "OpenNF: Enabling innovation in network function control," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 163–174, 2014.
- [13] J. Sherry *et al.*, "Rollback-recovery for middleboxes," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 227–240, 2015.
- [14] A. Gember-Jacobson and A. Akella, "Improving the safety, scalability, and efficiency of network function state transfers," in *Proc. ACM SIGCOMM Workshop Hot Topics Middleboxes Netw. Funct. Virtualization*, 2015, pp. 43–48.
- [15] S. G. Kulkarni, G. Liu, K. Ramakrishnan, M. Arumathurai, T. Wood, and X. Fu, "Reinforce: Achieving efficient failure resiliency for network function virtualization based services," in *Proc. ACM Conf. Emerg. Netw. Exp. Technol.*, 2018, pp. 41–53.
- [16] M. Z. Shafiq, L. Ji, A. X. Liu, J. Pang, and J. Wang, "A first look at cellular machine-to-machine traffic: Large scale measurement and characterization," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 1, pp. 65–76, 2012.
- [17] C. Mitchell, Y. Geng, and J. Li, "Using one-sided RDMA reads to build a fast, CPU-efficient key-value store," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 103–114.
- [18] P. Stuedi, A. Trivedi, and B. Metzler, "Wimpy nodes with 10GbE: Leveraging one-sided operations in soft-RDMA to boost memcached," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 347–353.
- [19] G. Kaur, M. Kumar, and M. Bala, "Comparing ethernet & soft RoCE over 1 gigabit ethernet," *Int. J. Comput. Sci. Inf. Technol.*, vol. 5, no. 1, pp. 323–327, 2014.
- [20] Click, "hotconfig." [Online]. Available: <https://github.com/kohler/click/wiki/Linuxmodule>. Accessed on: Mar. 14, 2020.
- [21] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst.*, 2015, pp. 5–16.
- [22] R. Laufer, M. Gallo, D. Perino, and A. Nandugudi, "Climb: Enabling network function composition with click middleboxes," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 4, pp. 17–22, 2016.
- [23] J. Schulz-Zander, S. Schmid, J. Kempf, R. Riggio, and A. Feldmann, "LegoFi the WiFi building blocks! The case for a modular WiFi architecture," in *Proc. Workshop Mobility Evol. Internet Archit.*, 2016, pp. 7–12.
- [24] D. Kirchner *et al.*, "Augustus: A CCN router for programmable networks," in *Proc. ACM Conf. Inf.-Centric Netw.*, 2016, pp. 31–39.



Heng Qi (Member, IEEE) received the bachelor's degree in mathematics from Hunan University, Changsha, China, in 2004, and the master's degree in software engineering and the Ph.D. degree in computer science and technology from the Dalian University of Technology, Dalian, China, in 2006 and 2012, respectively.

He was a JSPS Overseas Research Fellow with the Graduate School of Information Science, Nagoya University, Nagoya, Japan, from 2016 to 2017. He is currently an Associate Professor with the School of Computer Science and Technology, Dalian University of Technology. His research interests include computer networks and multimedia computing.



Keqiu Li (Senior Member, IEEE) received the bachelor's and master's degrees in mathematics from the Department of Applied Mathematics, Dalian University of Technology, Dalian, China, in 1994 and 1997, respectively, and the Ph.D. degree in computer science and technology from the Graduate School of Information Science, Japan, Advanced Institute of Science and Technology, Nomi, Japan, in 2005.

He has a two-year postdoctoral experience with the University of Tokyo, Tokyo, Japan. He is currently a Professor with the School of Computer Science and Technology, Dalian University of Technology. He has authored or coauthored more than 200 technical papers published in journals, such as the *IEEE/ACM TRANSACTIONS ON NETWORKING*, the *IEEE TRANSACTIONS ON MOBILE COMPUTING*, the *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, and the *ACM Transactions on Internet Technology*. His research interests include Internet technology, data center networks, cloud computing, and wireless networks.



Junxiao Wang received the B.S. degree in software engineering from Dalian Maritime University, Dalian, China, in 2014. He is currently working toward the Ph.D. degree with the School of Computer Science and Technology, Dalian University of Technology, Dalian.

His current research interests include software-defined networks, network function virtualization, and cloud computing.



Steve Uhlig received the Ph.D. degree in applied sciences with the University of Louvain, Louvain-la-Neuve, Belgium, in 2004.

He is currently a Professor of Networks with the Queen Mary University of London, London, U.K. His research interests include the large-scale behavior of the Internet, Internet measurements, software-defined networking, and content delivery.