Click-UP: Towards the Software Upgrade of Click based Modular Network Function

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

***Abstract*—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 work, 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 pipeline, letting separately managing logics for stateless operations and stateful operations become practical; (2) *essential module integration* - managing dependencies between modules, avoiding shipping unnecessary modules with neutral functionalities to target NF; (3) *local state migration* - migrating needed states seamlessly from old NF to 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 Virtualisation (NFV), 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].

The Click [4], thanks to its design, has been the best platform for the NF upgrade [5], [6], [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, Click based NF can be refactored by modifications to required module code rather than to the whole pipeline.

Junxiao Wang, Heng Qi, and Keqiu Li are with the School of Computer Science and Technology, Dalian University of Technology, P.R. China. Steve Uhlig is with the School of Electronic Engineering and Computer Science, Queen Mary University of London, UK. Heng Qi is the corresponding author (hengqi@dlut.edu.cn).

Despite these features, its internal architecture has unfortunately not kept up with some specific problems of the software upgrade. We conclude the limitations[1] 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 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 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], [11], [12]. State migration from remote, however, takes time, inherently increases the upgrade latency and ignores the local state update during the checkpointing. (2) logging all inputs (i.e., packets) and using deterministic replay in order to rebuild the state [13], [14], [15]. The solution of this kind, in fact, works at the cost of a substantial 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 with 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,

[1]Note 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.

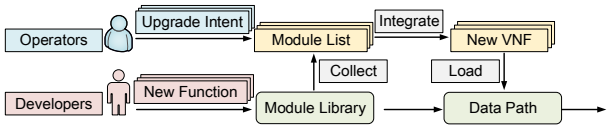


Fig. 1. Lifecycle for the software upgrade of Click based modular NF.

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 target NF. With managing the functional dependencies between modules, the way modules compiled and linked sticks to the dependencies, eliminating the burden involved in redundantly shipping inessential modules with neutral functionalities, and significantly cutting down the upgrade latency as well.

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 old NF, and are seamlessly reloaded to target NF through the push interface. Access to external/shared states is also migrated. In doing so, needed states are migrated effectively in local, and significantly eliminating the service disruption.

To demonstrate the efficiency of our solution, we present two typical NFs implemented on top of Click-UP: NAT (network address translation) 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 paper is organized as follows. §-II presents the motivation and design of the modular state abstraction. Followed by §-III, we show how to compile/link the essential modules according to the managed dependencies. §-IV presents the migration of needed states between the old instance and the target instance. In §-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 §-VI. In §-VII, the related work is summarized. Conclusions are provided in §-VIII.

II. Modular State Abstraction

A. Motivation and Challenges

By rethinking the software upgrade of Click based modular NF, we denote its lifecycle with Figure-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; by collecting the 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 not only include the abstractions 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.

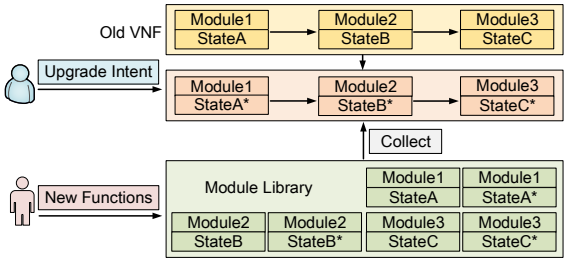


Fig. 2. The traditional software upgrade, where the target NF is organized in a stateless way (i.e., lacking of modular state abstraction).

As shown in Figure-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 non-modular 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 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 dependency 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 pipeline. As illustrated in Figure-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 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. Please note that the meaning of 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 Figure-4, most NF code can be logically divided into three basic parts: initialization, packet receive loop, and packet processing. The initialization code runs when

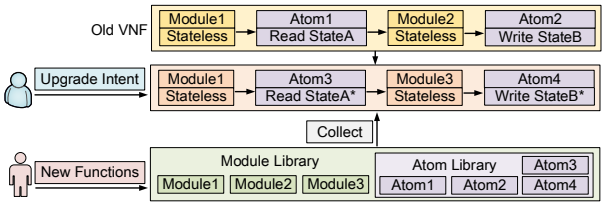


Fig. 3. The software upgrade with Click-UP, where the target NF is organized in a stateful way.

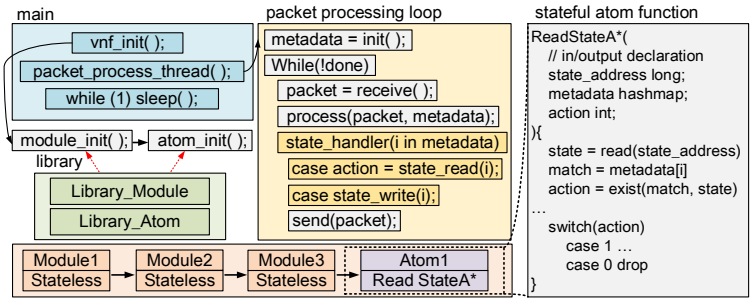


Fig. 4. The processing of a NF using the modular state abstraction.

the NF starts. It reads and parses configuration input, loads modules or files. All of these can be done in the `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 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 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.

VNFs	State	Key	Value
VPN	Authorised Token	5-Tuple	Token String
Firewall	Whitelist	Cluster ID	5-Tuple
	TCP FSM	5-Tuple	FSM State
NAT	Pool of IPs and Ports	Cluster ID	IP and Port List
	Mapping of Port	5-Tuple	IP and Port
IPS	Automata State	5-Tuple	State Number
	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

TABLE I
DECOUPLED STATES IN THE ATOM OPERATION FOR TYPICAL NFS.

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:

The load balancer. Upon receiving a tcp connection request, 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, atom of ReadState retrieves the assigned server for that flow and forwards the packet to the server.

The signature based IPS. Upon receiving a new flow, 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, 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. Towards the tcp out-of-order problem, atom of WriteState buffers out-of-order packets into the data store. The atom of ReadState retrieves buffer to reassemble the stream of bytes.

The NAT. Upon receiving a tcp connection request, atom of WriteState retrieves the list of ips and ports from data store 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, atom of ReadState retrieves the assigned pair of (ip, port) for that flow, and updates the packet header.

The VPN. Upon receiving a tcp connection request, 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.

The whitelist based firewall. Upon receiving a new flow, 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, 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 Figure-5). 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 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 the modular state abstrac-

```
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. The stateful target NF of a whitelist based firewall. The text corresponds to a directed acyclic graph (DAG) using stateless modules and stateful atom operations as nodes.

tions in Click-UP are general for more NFs and have the efficiency of their implementation, 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 §-III, we show how to integrate the modules and atoms determined by the target NF. In §-IV, we introduce how to migrate needed states (related to the atoms) from the old NF to the deployed new NF.

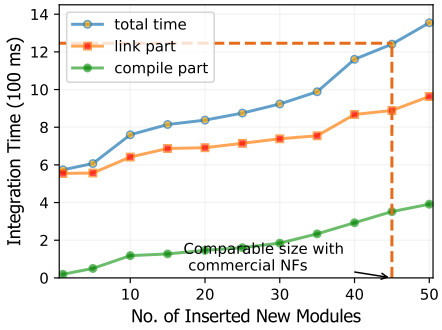


Fig. 6. Integration time of the software upgrade in Click. The data is obtained in a server with Intel Xeon (E5-2630v3) 64GB RAM.

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 SLA may require that outdated NF instances never process traffic for more than 10 minutes 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; (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 will lead to an 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` modules[2] inserted into the original pipeline. The integration time of the target NAT was measured by observing the timestamp of the first printed flow. We plotted in Figure-6 the average over 20 experiments.

We find 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. Towards 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.

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

B. Design and Implementations

To cut down unnecessary burden involved by module integration, we propose a more lightweight integration 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 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 Figure-7, the components of the integration layer are *Intent Resolution Component* (IRC), *Dependency Management Component* (DMC), *Configuration Management Component* (CMC) and *Compile Link Component* (CLC).

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

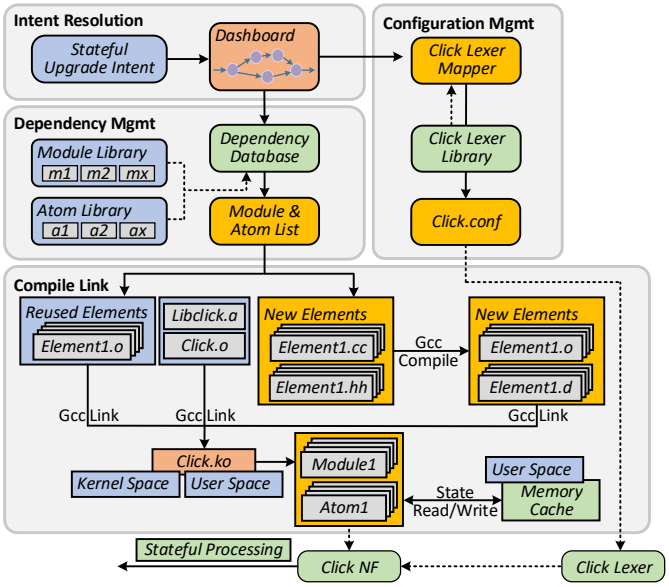


Fig. 7. Design of lightweight integration layer in Click-UP.

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

Dependency Management Component. DMC is with a dependency database, which stores the module relationship used by dependency exploration. Should know, in most cases, the declared modules in target NF are not equal to the required modules in executable NF. 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 dependency model as follows:

We have two classes of dependencies defined in the modularity of Click, element[3]-to-functionality and functionality-to-functionality. When an element depends on a functionality without itself providing functionalities, we refer to the dependency as an element-to-functionality dependency. When the element provides functionalities and requires other functionalities to provide its own functionality, we refer to the dependency as a functionality-to-functionality dependency.

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 dependency exploration is with all the declared element instances of target NF in a valid stage.

Compile Link Component. CLC is with a re-designed element compiler and linker. Note that we implement both of stateless module and stateful atom with unified Click modular files `Element.cc` and `Element.hh`. Therefore, input of CLC (i.e., output of DMC) is a set of Click elements. We first categorize these elements into two boxes: (1) elements used by both of old NF and target NF; and (2) elements only used by 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 userspace, respectively. In both of two cases, the atoms in the target NF will read/write the needed states (via the stateful operations defined in §-II) within a memory cache of userspace. In §-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 (IDS), address mappings in a network address translator (NAT), or server mappings in a stateful load balancer.

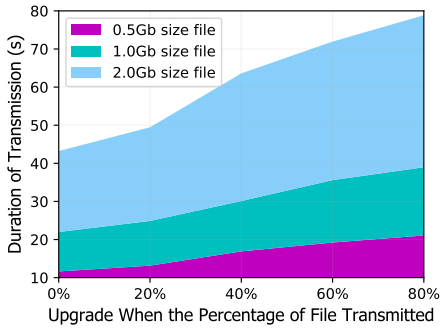


Fig. 8. The transmission time with the state loss. The data is obtained in an http server with 1.0Gbps NIC.

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 0.5GB, 1.0GB and 2.0GB 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 Figure-8, the transmission duration (y-axis) is based on the average across 20 experiments.

[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.

We find 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 11s for the 0.5GB size file and 43s for the 2.0GB file. When the upgrade happened after 40% of the file has been transmitted, the total average transmission time became 16s and 63s, 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 Click's state migration during the software upgrade.

B. Design and Implementations

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

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.

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.

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 target NF transparently. The states of 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 on a unique structure of NF to enable state migration at the local memory. Figure-13 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:

State Management Component. 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 play a role responding calls from pull/push interface, fetching or embedding needed states during upgrade.

Buffer Management Component. 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 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 too small size, 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. BMC can responde calls from pull/push interface, fetching or embedding packet buffers during upgrade.

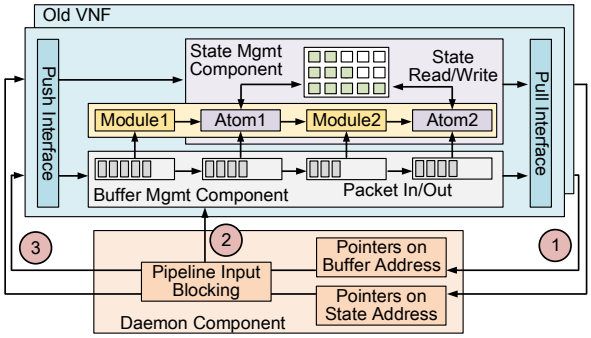


Fig. 9. High-level design of state migration in Click-UP.

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

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

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.

Rollback. Click-UP does compose, compile and link target click.ko offline (i.e., without disrupting services during this period), then unload the old click.ko and reload the new one. To rollback from failures (e.g., the target NF startup failed), we employ a status checking, which checks the bootstrap status of 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.

SMC and all the states are resided in Linux userspace for

security and scalability. Remember that Click-UP supports pipeline running both in kernel and user space. In the case that pipeline is running in kernel space, we leverage the Netlink socket for inter-process communication (IPC) both between the kernel and userspace processes, and between different userspace processes. In the case that pipeline is running in userspace, 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 userspace mode). This is due to its fast deployment, low-performance overhead, and high reusability. The old and target NF are implemented as Docker instances with independent cores and memory space/region. In doing so, we ensure that switching pipeline does not affect each other. For network connectivity, we share the physical interface among each of the containers (pipelines). Towards 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? (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

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

NAT: is based on Click Mazu NAT, a modular network address translation implemented by Mazu Networks, and commonly used in academic research.

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

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-II 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.

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%)

TABLE II
THE SOFTWARE UPGRADES IN THE SIMULATIONS.

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

TABLE III
THE NUMBER OF MODULES AND ATOMS IN THE EXPERIMENTAL NFS.

Topology. We evaluated the performance of software upgrade using a topology shown in Figure-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 GRE tunnels by creating VTEPs on the nodes of http client, http server, and Click NF. All the upgrades happen during the procedure of the file streams.

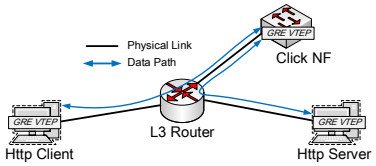


Fig. 10. Topology of the simulations. The data path is implemented with the GRE technique.

Devices. We consider two dominating scenarios to deploy NFs: telecom network and edge network, whose features are distinct. The devices in 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. Figure-11 shows our testbed environment in simulated scenario of edge network. Please find the demo video at https://www.youtube.com/watch?v=5G244I0LEYg.

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

TABLE IV
THE DEVICES IN THE SIMULATIONS.

Schemes Compared. In terms of module integration, we selected integration scheme of native Click as the baseline.

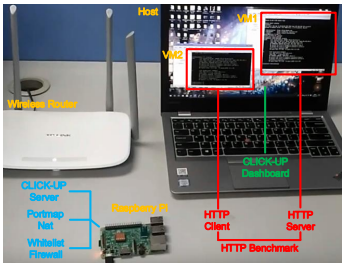
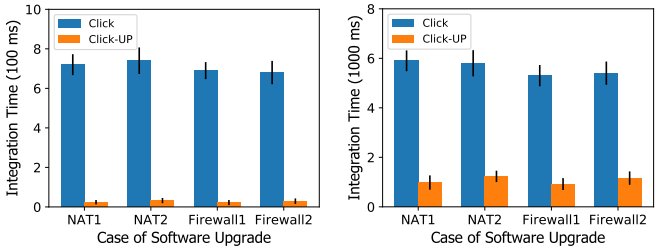


Fig. 11. The deployment environment in simulated scenario of edge network.



(a) TN. (b) EN.

Fig. 12. The comparison of integration time between Click-UP and native Click in two simulated scenarios.

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], [11], [12]. When upgrade happens, old NF first sends its states to a remote instance, followed by target NF getting back needed states from the same remote instance. (2) *Packet replay* [13], [14], [15]. When upgrade happens, old NF first sends its states to a remote instance, then let the packets arriving later be logged; after 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

Integration Time. the time for finishing the building job of target NF. 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.

Service disruption Time. the time topped up by state inconsistencies. With the inconsistent states, 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 pre-defined 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, 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.

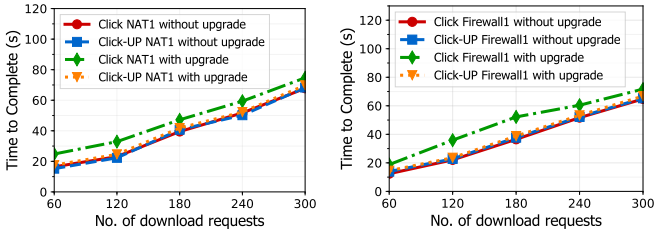
With the measured integration time, Figure-12(a) and Figure-12(b) plotted a comparison between Click-UP and native Click under two simulated scenarios, where the case of upgrade varies in the x-axis. We made the following observations: (1) Compared with the native version, Click-UP has a significantly reduced integration time. This is expected since Click-UP leverages the dependency 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, that of TN has significantly less integration time (26 ms vs. 941 ms) and higher reduction rate (96.1% vs. 78.2%). The reason is, powerful computation performance (from e.g., CPU and RAM) speeds up the process of compiling and linking, thus boosts 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 §-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 NAT on top of native Click and Click-UP (1) without software upgrade; (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.

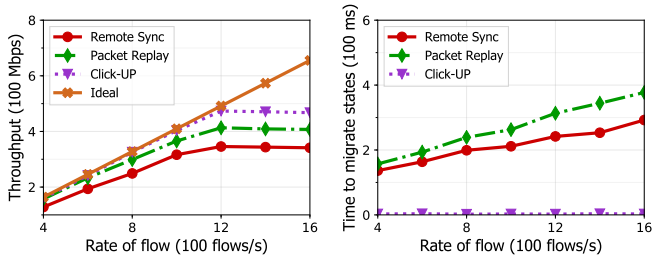
We conduct the experiment using the simulated scenario TN. Figure-13(a) and Figure-13(b) shows our results where we download up to 300 10MB files in a loop of 60 concurrent http downloads through the firewall and the NAT. Figure-13(a) provides a comparison of time taken to satisfy completed requests 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 are wasted. In contrast, there is almost no service disruption time for Click-UP.

Similarly, as plotted in Figure-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



(a) Click NAT1 and Click-UP NAT1. (b) Click Firewall1 and Click-UP Firewall1.

Fig. 13. The comparison of time taken to satisfy completed requests between Click-UP and native Click.



(a) comparison of throughput. (b) comparison of migration time.

Fig. 14. The comparison of throughput and migration between our local migration schemes and other baseline schemes.

re-initiating the connections after a tcp connection timeout. We find the firewall of Click-UP is unaffected due to the seamless state migration and almost match the download time does not experience 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.

Should know, the migration time and disruption time heavily depends on the flow rate, because the flow rate have a significant impact on the number of stored states. We experimented using the simulated scenario TN. Figure-14(a) and Figure-14(b) plotted a comparison between 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. Figure-14(a) provided the measured throughput during the software upgrade. The four curves in this figure represent: the ideal throughput (Ideal) which 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 to migrate 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 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 grows 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, 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 grows up, the number of associated states will increase accordingly, and the gap in 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.

Figure-14(b) plotted a comparison of time to migrate states between Click-UP and other schemes. Observe that Click-UP always outperforms other schemes, and the migration time of Click-UP has no significant change with increasing flow rate. This owing to the local pointer copy instead of remote data copy, and whose consumed time is independent with the rate of flow and the number of states. Compared to Click-UP, remote synchronization and packet replay both depends 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 to 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 RDMA [17], i.e., SoftiWARP [18] and SoftRoCE [19]. Figure-15 plotted 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, a gap still is there compared to the performance in local.

Back to Figure-14(b), we also observed that the migration time of packet replay is several ms 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 Figure-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 does have 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 need 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, travel through a firewall, reach the server, and sent back to the client. The client records the sending time and receiving time of every packet to compute the packet RTT. Figure-16(a) and Figure-16(b) showed the RTT of 10 flows traversing the firewall under two simulated scenarios. We measured the average over 20 tries.

Figure-16(a) plotted a comparison under the simulated scenario TN. Observe that the extra RTT caused by Click-UP is less than P4-bmv2 (3.2% vs. 7.7% on average). Figure-16(b) plotted a comparison under the simulated scenario EN, and a similar observation can be found under EN (8.9% vs. 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

Distributed Deployment. In this paper, we assume that the target NF and 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 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 paper 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.

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 which 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: firstly, should find the stateful operation from pipeline and replace it with correspondent Click-UP atom (as illustrated in §-II); secondly, remove the stateful operation from pipeline and build the remaining stateless operation as a Click module; thirdly, let a mapping relationship set up between them. The module redistribution may in fact take a 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.

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 orginal 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 pose 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, above impact can be ignored. We further demonstrate Click-UP's integration performance in §-V.

VII. Related Work

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

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. Firstly, 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. Secondly, hotconfig, although shifts packets in queues from the old NF to the new one; it does not deal with internal network states. In contrast, Click-UP employs dependency management to considerably cut down the overhead of reintegration and provides a seamless state migration.

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 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 who 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.

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 5MB, fast boot, high-performance, Click-driven virtualized

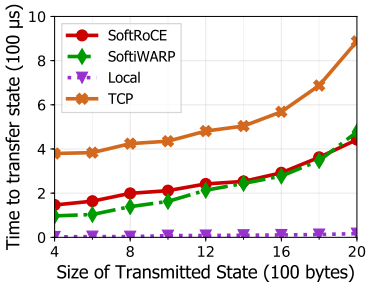
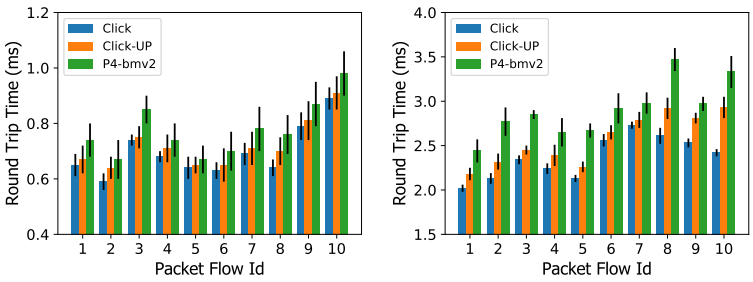


Fig. 15. The comparison of state transmission delay.



(a) TN. (b) EN.

Fig. 16. The comparison of data plane latency between Click-UP, native Click and P4-bmv2 in two simulated scenarios.

software middlebox platform on a commodity server. The fastClick [21] system exploits netmap and dpdk to leverage the power of hardware multi-queues, multi-core processors and non-uniform 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 paper, we investigated the limitations of 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 network functions. Based on native Click, we made three main improvements in Click-UP.

Firstly, 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. Secondly, we achieved the essential module integration with the management of functional dependencies. The burden involved in redundantly shipping inessential modules with neutral functionalities are eliminated. The experiment results showed that the integration time is significantly cut down. Thirdly, we achieved the state migration scheme at local memory. The internal states are collected from old NF and reloaded into new NF with seamlessness. The experiment results showed that there is almost no service disruption time with Click-UP.

Acknowledgment

This work was supported by the State Key Program of National Natural Science of China (Grant No. 61432002), NSFC Grant Nos. 61772112, U1701263, 61672379, and 61751203, and the Science Innovation Foundation of Dalian under Grant 2019J12GX037.

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 Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[2] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[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 Systems Journal*, vol. 13, no. 4, pp. 3877–3888, 2018.

[4] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, 2000.

[5] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming slick network functions," in *Proceedings of ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015.

[6] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: a software-defined framework for developing, deploying, and managing network functions," in *Proceedings of ACM SIGCOMM*, 2016.

[7] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware," in *Proceedings of ACM SIGCOMM*, 2016.

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

[9] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[10] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Proceedings of the Annual Symposium on Cloud Computing*, 2013.

[11] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[12] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 163–174, 2014.

[13] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo *et al.*, "Rollback-recovery for middleboxes," *ACM SIGCOMM Computer Communication Review*, 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 *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, 2015.

[15] S. G. Kulkarni, G. Liu, K. Ramakrishnan, M. Arumaithurai, T. Wood, and X. Fu, "Reinforce: achieving efficient failure resiliency for network function virtualization based services," in *Proceedings of ACM Conference on emerging Networking EXperiments and Technologies*, 2018.

[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 performance evaluation review*, 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 *Proceedings of USENIX Annual Technical Conference*, 2013.

[18] P. Stuedi, A. Trivedi, and B. Metzler, "Wimpy nodes with 10gbe: leveraging one-sided operations in soft-rdma to boost memcached," in *Proceedings of USENIX Annual Technical Conference*, 2012.

[19] G. Kaur, M. Kumar, and M. Bala, "Comparing ethernet & soft roce over 1 gigabit ethernet," *International Journal of Computer Science & Information Technolo*, vol. 5, no. 1, pp. 323–327, 2014.

[20] Click, "hotconfig," https://github.com/kohler/click/wiki/Linuxmodule.

[21] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2015.

[22] R. Laufer, M. Gallo, D. Perino, and A. Nandugudi, "Climb: Enabling network function composition with click middleboxes," *ACM SIGCOMM Computer Communication Review*, 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 *Proceedings of the Workshop on Mobility in the Evolving Internet Architecture*, 2016.

[24] D. Kirchner, R. Ferdous, R. L. Cigno, L. Maccari, M. Gallo, D. Perino, and L. Saino, "Augustus: a ccn router for programmable networks," in *Proceedings of ACM Conference on Information-Centric Networking*, 2016.