A Highly Reliable and Programmable Software Defined Coalition (SDC) Architecture using Multiple Control Plane Composition with Distributed Verification

Kerim Gokarslan*, Geng Li*, Patrick Baker†, Franck Le§,
Sastry Kompella‡, Kelvin M. Marcus¶, Vinod K. Mishra¶, Jeremy Tucker†, Y. Richard Yang*, Paul Yu¶
*Yale University, USA, †UK Defence Science and Technology Laboratory, United Kingdom
‡U.S. Naval Research Lab, USA, §IBM T.J. Watson Research Center, USA, ¶U.S. Army Research Lab, USA

***Abstract*—A major potential benefit of software-defined SDx architectures is agility achieved through logically centralized programmability. The dependency on a centralized control plane (CP), however, can have substantial reliability issues, when the CP can be weakly connected to the data plane (DP), or even disconnected from the DP. Distributed CPs such as MANET (e.g., OLSRv2) can be more reliable but are less programmable and cannot provide global policy guarantees. In this work, we design *Carbide*, a novel, fundamental control architecture for SDC that achieves a high level of reliability and programmability through the composition of multiple control planes. Introducing both a novel distributed verification framework and a general resource management framework, *Carbide* selects CPs intelligently and in distributed manner such that they satisfy global constraints. We conduct data-driven evaluations in military settings using EMANE, and demonstrate that *Carbide* can achieve average downtime that is only 1/3 of the most reliable individual CPs. This work provides a highly reliable, verifiable, secure, and flexible CP design tailored for SDC. Preliminary results appeared in DAIS 2019.**

I. Introduction

The architecture of a control plane determines key properties of a network such as robustness, flexibility, and efficiency, which are vital to the network's behavior. As a result, substantial efforts have been devoted to design ideal control plans and a variety of control planes have been proposed. Unfortunately, each control plane designed so far still has both benefits and drawbacks. For example, traditional control planes (*e.g.*, OLSRv2 [5], DSDV [16]) are distributed control planes, and hence can be highly robust thanks to their distributed design; but they are often rigid and limited in flexibility due to fundamental feasibility constraints in distributed computing.

The emerging software-defined architectures allow centralized control to avoid complex distributed consensus and hence may achieve better flexibility, but they can face substantial robustness challenges when their controllers are disconnected or weakly-connected from the data planes. In particular, such challenges become a significant drawback in software-defined coalition (SDC) networks with high mobility and dynamicity, lossy controller and disrupted asynchronous communication channels [23]. For example, for the in-band control plane, the control messages may have a race condition with data messages, or the connection with the controller may get lost, resulting in unreliability and huge latency. Deployment of multiple controllers in software defined networking (SDN) architectures for robustness is straightforward but also limited in response to network partition, as it is impossible to deploy a controller in every "potential" partitions.

When the drawback of a given control plane is not acceptable for an otherwise highly desirable control plane, for example, when the robustness of OLSRv2 is highly desirable, but the risk that OLSRv2 cannot ensure properties such as ACL or waypoints, a natural approach is to use multiple control planes. In particular, the authors of [21], [22] propose a hybrid SDN architecture, in which an SDN control plane serves as the primary control plane, and a traditional routing protocol (*e.g.*, OLSRv2) is also deployed to serve as the backup. When the SDN controller is no longer available at a local node, the node switches to use the routes computed by the traditional protocol.

As beneficial as the hybrid SDN architecture is, by allowing the reuse of substantial control planes, it may introduce serious issues when the switching of control planes is *ad hoc*. First, consider generic safety issues. Assume that neither the SDN control plane nor the traditional protocol has routing loops. Consider a possible ad-hoc switching setting where one node switches from SDN to the traditional protocol, but a downstream node may still be using SDN. Highly undesirable routing loops can then emerge in such a setting. Next, consider security issues. Consider a simple example where a network has an essential requirement that certain traffic be dropped and the SDN control plane has been correctly configured to enforce this requirement. The traditional protocol, however, may not be correctly configured or may not even have this filtering capability. When a node switches from SDN to the traditional protocol, the requirement is not enforced and a security breach is introduced.

In this paper, we explore a novel network control architecture which we call the *Carbide* architecture. Similar to the hybrid SDN architecture, our architecture allows the modular reuse of multiple, existing, substantial control planes. More general than the hybrid SDN architecture, *Carbide* is not limited to one SDN as primary and one traditional protocol as a backup; instead, our architecture allows a generic set of control planes, in flexible orders. Different from ad-hoc hybrid SDN, our architecture introduces a novel, fundamental paradigm which we call *consistent composition* of multiple

control planes. By consistent, we mean that a network operator can enforce that the routes composed are self consistent, so that safety issues such as the aforementioned loops are not introduced by using multiple control planes; further, by consistent, we also mean that the behaviors of the composed control planes are consistent with user requirements, so that security issues such as the aforementioned security breaches are not introduced. Since our goal is similar to the design of alloys, which composes multiple metals to obtain individual strength and avoid individual weakness, we name our architecture the *Carbide* architecture.

Realizing the *Carbide* architecture is not trivial. One fundamental challenge is: How can each network device know which control plane to use to achieve network-wide correctness? Network verification appears as a potential tool and has received substantial attention lately (*e.g.*, [4], [8], [9], [12]–[14], [26]), but the existing work is not designed for control plane composition (*e.g.*, using a centralized design).

In this paper, we explore distributed verification as a novel, core technique to address the aforementioned challenge and realize the *Carbide* architecture. By distributing the selection of control planes at individual nodes, in real-time, at fine-grain (different packets may use different control planes), according to rigorous, distributed verification, our design avoids any single point of failure and still ensures paths to be correctness-compliant. By abstracting each control plane at each device as a blackbox and taking its output as a virtual forwarding information base (vFIB), we achieve generic composition.

We instantiate the *Carbide* system and evaluate it using experiments on large-scale simulations. We show that by composing multiple control planes in real time, *Carbide* takes the advantages of both centralized and distributed control planes. Specifically, the system experiment results demonstrate that in the event of a link failure, *Carbide* reduces downtime by up to 65% compared to SDN, and by up to 77% when compared to distributed control planes. We also demonstrate the performance of *Carbide*'s distributed verification module on the NATO IST-124 Anglova scenario [19] using Extendable Mobile Ad-hoc Network Emulator (EMANE) [18] to emulate our mobile network.

The highly promising preliminary results motivate us to introduce additional capabilities in *Carbide*: (1) a BGP-inspired damping capability to go from passive switching to more active control to improve on stability; (2) a light-weight packet tag (CVV) capability to prevent forwarding loops across control planes; (3) an adaptive resource control capability to intelligently adjust and allocate resources (*e.g.*, CPU, memory, bandwidth) to control planes and the verification processes; (4) a novel optimized verification messaging protocol to minimize broadcasting overhead; and (5) a predictive mobility tracking model to improve system agility in highly dynamic SDC networks.

II. Overview

In this section, we present a high-level overview and the typical workflow of *Carbide*. The key components of *Carbide* are shown in Figure 1. Each network device runs (1) a control plane layer, (2) a novel online composition layer, and (3) an enhanced data plane layer.

Control Plane Layer. The control plane layer consists of a set of control plane (CP) instances $\mathcal{CP} = \{CP_1, ..., CP_k\}$ running simultaneously. A CP instance may be centralized (*e.g.*, SDN) or distributed (*e.g.*, OLSRv2). For example, a device may run

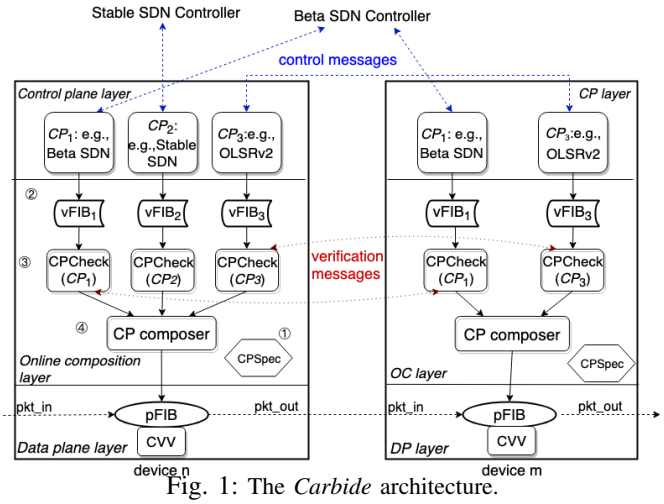


Fig. 1: The *Carbide* architecture.

three CP instances: CP_1 and CP_2 as two SDN CPs receiving OpenFlow messages from a new release and a stable release of SDN controller, respectively, and CP_3 as a traditional link-state CP such as OLSRv2. *Carbide* treats every CP instance as a black-box and only depends on the output (*i.e.*, forwarding information) of each CP instance. This design decision allows *Carbide* to re-use any existing implementation (open source or commercial) for each CP instance.

Online Composition Layer. This novel layer dynamically composes the information from the CP instances to satisfy the network-wide requirements specified by operators. This layer consists of four components.

- ***CPSpec***: This component enables operators to specify desired correctness requirements (*e.g.*, waypoint traversal, ACL) of different traffic types, and the overall preference order among CP instances.
- **vFIB**: Each CP is associated with a virtual forwarding information base that collects and stores the forwarding information from that CP instance.
- ***CPCheck***: Each CP is also associated with a light-weight event-driven verification module called *CPCheck*. Upon detecting any change in a vFIB or local forwarding state (*e.g.*, port/link failures), *CPCheck* verifies the correctness requirements for that CP in real time. The goal of *CPCheck* is to detect which packets have the desired correctness requirements satisfied. One key challenge for *CPCheck* is that it cannot rely on a centralized entity, as the centralized entity may not be reachable in the case of network partitions, and it is hard to collect all the distributed configuration file at one place. Details of the verification module are described in §III.
- **CP composer:** The CP composer takes as inputs the forwarding rules from the vFIBs and the verification results from the *CPCheck* modules and computes the configuration to enforce at the data plane layer.

Implementing *Carbide* on topologies with certain features, *i.e.*, highly mobile and wireless environment in the SDC networks, presents certain key challenges. In *Carbide*, we implement multiple strategies that lets us tackle such problems and optimize the performance of a multi-control plane system.

Optimized Flooding. We propose a novel approach for disseminating information, such as updates in *Carbide*'s policies, quickly and while spending very little bandwidth. The key idea

is to broadcast the message along paths that only go through the routers that need a particular policy update. For instance, if a packet is dropped due to a link failure and one of the routers in the path realizes this failure, it will send out the *Carbide* verification message. We discuss this idea in a greater detail in Section IV-A. All the routers that are informed about the failure using this message will update their verification results.

Mobility Tracking. *Carbide* in a wireless environment uses mobility tracking to keep the routing policies updated, which helps us minimize packet losses and increase response time quickly. This approach is especially beneficial in highly mobility scenarios such as the tactical edge. Mobility tracking is discussed in Section IV-B.

Data Plane Layer. This layer provides two main functions:

- Process data packets at line rate based on rules installed by the CP composer in the device's physical FIB (pFIB).
- Prevent forwarding loops for data packets by adding a control plane visiting vector (CVV) tag in each packet to record the previously visited control planes.

Requirement Specification. The aforementioned *CPSpec* is for each device and generated from a global requirement specification. Specifically, the global specification consists of a sequence of requirement items, with each item specified as a $(prop_j, DHS_j)$ pair, where $prop_j$ is a path property and DHS_j the desired header subspace for $prop_j$. A path property is expressed as a match on path hops, using a regular expression grammar inspired by Flowexp [12]. An example of a requirement item is to ensure path *reachability* property for header subspace of *all traffic except on port 23*, written as $(\texttt{.*>\$}, \{p \mid p.dstPort \neq 23\})$; another example is *All traffic with destination D must pass through a load balancer at device M*, written as $(\texttt{.*[d=M].*\$}, \{p \mid p.dstIP = D\})$. Each $(prop_j, DHS_j)$ pair will be localized to the *CPSpec* of corresponding devices as described in §III-C.

Adaptive Resource Control. *Carbide* consists of a large number of running processes (*e.g.*, control plane instances, verification modules, CP composer) that compete for shared resources (*e.g.*, CPU, memory, bandwidth). Resource control, therefore, plays a critical role. Without resource control, a control plane process may use all of the device's resources, resulting in resource starvation for other processes, and ultimately causing poor overall performance. To address this issue, we develop an adaptive resource control mechanism to adjust and allocate resources to different processes. Intuitively, when several of the preferred control plane instances already achieve the desired objectives and performance, the resources allocated to other control plane instances (*e.g.*, CPU and control channel bandwidth) can be reduced. The details of the resource control will be described in Section IV-C.

Workflow. We now present the workflow to illustrate how it works in *Carbide*. (1) Through the *CPSpec* component, operators specify the global requirements, and the CP preference order. (2) As the CPs run, they gradually populate their vFIBs with forwarding rules. (3) As the vFIB of each CP_i evolves, the associated *CPCheck* module verifies CP_i and outputs the set of packets for which CP_i fails to satisfy the requirements. (4) The CP composer takes results from each *CPCheck*, and the vFIB contents as inputs to configure the data plane. Specifically, the CP composer creates rules for packets to be processed by the most preferred CP whose verification result returns true. For example, when there is a data plane change (*e.g.*, rule updates or link failures) for CP_i, the *CPCheck* of CP_i at the closest device will update its verification results for the affected packets. While the unaffected packets continue to use CP_i, *Carbide* can immediately switch to another available CP for the affected packets. If no CP is available (*e.g.*, a link failure affecting all CPs), *Carbide* waits for all CPs to recompute and chooses the first CP that returns a new valid route. Hence, *Carbide* always achieves the minimum convergence time among all CPs.

III. Distributed Verification

How can a device locally verify that a route from a CP satisfies the desired path properties (*e.g.*, waypoint routing, disjoint paths)? This question is essential in *Carbide* to enable devices to locally decide which CP to select to forward a packet. This section presents *CPCheck*, a novel distributed algorithm associated with each CP, to address this problem. The main idea consists in having each switch store not only the next-hop but the downstream path through an efficient lightweight data structure in §III-A, which *CPCheck* can then use to verify whether the path is consistent with the desired requirements. One challenge lies in ensuring that the path information is correct and up-to-date. To ensure it, we introduce an efficient distributed update protocol in §III-B to update the information at relevant switches in response to data plane updates (*e.g.*, failures, configuration changes). §III-C describes how using this information, devices can locally verify the compliance of the available routes with the desired objectives, and select the most preferred ones. §III-D then describes how *Carbide* uses link metric values to update equivalence class tables. As each CP is associated with an independent *CPCheck* instance, we focus on one single CP and its associated *CPCheck* for the rest of this section.

A. Local Equivalence Class (LEC) Tables

At each device, *CPCheck* consists of local data structures known as Local Equivalence Class Tables (LEC Tables) which store the relevant forwarding behavior for outgoing flows at each device. To motivate this design, we first consider a naive distributed implementation of existing centralized verification systems (*e.g.*, VeriFlow [14], NetPlumber [12]) by replicating the global data structures on each device. Devices would maintain the data structures by broadcasting any local flow rule updates across the network, and each device would individually compute correctness as specified by the centralized verification system. However, this approach has two major limitations: (1) the memory requirements of these global data structures generally scale with the *total* forwarding rule space of the *entire* network, which does not scale well in larger networks, and (2) network devices often have much less computing power than a typical controller, which may hurt verification time.

To avoid these limitations, we observe that each device only needs to be aware of forwarding rules at subsequent devices that affect its outgoing flows. We can then trim the stored data by only considering the subset of network devices reachable by downstream flows. To illustrate these insights, consider Figure 2. Any forwarding rules at device E or device F will not affect outgoing flows from device C, so device C need not be aware of such forwarding rules. To take advantage of this, we will introduce the notion of Local Equivalence Classes (LECs).

LECs. At each device, *CPCheck* partitions the packet header space into subsets, called LECs, such that only packets whose headers are in the same LEC use the same *unique* downstream forwarding path. This associative map between LECs

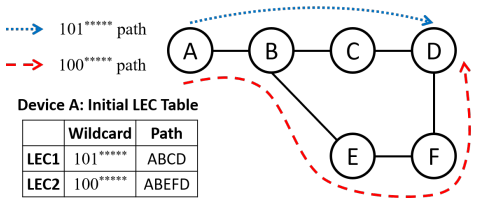


Fig. 2: An example with two flows using $ABCD$ and $ABEFD$.

and forwarding paths will form LEC Tables, the core data structures at each device, and this data will provide a complete local context which *CPCheck* can use to query for general correctness requirements. Figure 2 shows the initial LEC table stored at device A. Note that devices C, E and F only have one LEC.

To initialize the data structures with correct LEC information, we utilize a vector-based algorithm to aggregate next-hop information across devices.

B. Distributed Real-time Updates

We now describe the distributed algorithm to update the LEC tables across switches in response to real-time data plane updates. We first introduce the two messages *CPCheck* relies on. Then, we illustrate how using these two messages, the LEC tables are updated across switches in response to forwarding rules being added, modified, or deleted, and to other network events (*e.g.*, link failures). The full algorithm is omitted for space considerations.

LEC Path Update. The goal of this message is to announce changes in LEC that may affect other switches. More specifically, upon locally detecting a change (*e.g.*, forwarding rule update), or a network event (*e.g.*, link failure), *CPCheck* generates an *LEC path update message*, (n, HS_{aff}, P), where n is the device at which the network event was detected, HS_{aff} is the header subspace whose behavior is affected by the update, and P is the new path taken by HS_{aff} starting at device n. This announcement is broadcast to all devices in the network Then, when receiving this announcement, each device performs a local search to identify affected LECs and updates their path information. To achieve this, the device checks for all LECs with a *path dependency* on n, or in other words, all LECs whose forwarding paths contain n. In addition, the device checks for LECs with a *header space dependency* on HS_{aff}, or in other words, LECs that overlap with HS_{aff}. For each LEC that has both path and header space dependencies on the incoming announcement, the device then updates its path according to the new segment P.

LEC Poll. The goal of this message is for a switch to retrieve LEC forwarding information from other nodes. More precisely, in some instances such as when a forwarding rule is added or modified, a device may not have the needed downstream path P stored in its LEC table. In such case, the device sends an *LEC poll message* to the next-hop n' of the new rule to request the relevant path. In response, device n' floods its information to the network directly.

We now consider the topology shown in Figure 2 to illustrate the behaviors of *CPCheck* in response to various network events. In particular, we describe in details the actions when a forwarding rule is modified (U1) and deleted (U2). The behavior when a new forwarding rule is added and in response to link failures are very similar to U1 and U2 respectively and are not described in detail due to space limitation.

Modification of a Forwarding Rule (U1). As shown in Figure 3(a), when device B receives its new forwarding rule $< 101^{*****}$, Next-hop: $E >$, B does not have the downstream routing information for 101^{*****}. As such, B first sends an LEC poll to the rule's specified next-hop, *i.e.*, E. In response, E calculates the new path segment for 101^{*****} and broadcasts the LEC path update $(B, 101^{*****}, BEFD)$ to all devices in the network. Upon receiving this update, B notes that the device specified in the incoming update message is itself and updates its LEC table with the entry $(101^{*****}, BEFD)$. Similarly, A updates its LEC table. C, E and F do not update their LEC tables because they do not have any path dependency on B. The distributed verification protocol can converge quickly because 1) it utilizes a link-state-based update algorithm allowing nodes (*e.g.*, A and B) to update their LEC tables in parallel, and 2) given a forwarding rule modification, only one broadcast is needed: *e.g.*, after E broadcasts the changes, even though B updates its LEC table, B does need not to broadcast its update. As a result, nodes (*e.g.*, A) will not receive and need to process multiple broadcasts. Adding a forwarding rule follows the same workflow.

Deletion of a Forwarding Rule (U2). Figure 3(b) shows the deletion of a flow rule at C. On detecting the flow rule deletion, C broadcasts a LEC path update $(C, 10^{******}, NULL)$ across the network. Since A and B have dependencies on both 10^{******} and C, they update the segments after C in their affected LEC entries (101^{*****}). In contrast, other devices do not update their LEC entries because although they may have header space dependencies, they do not have any path dependency. Similar workflow applies when a device detects a local failure (*e.g.*, link $C - D$ failure.)

Concurrency of Flow Rule Updates (U1 and U2). Previous examples only considered flow rule updates (*e.g.*, U1 and U2) separately. In reality, multiple updates (*e.g.*, U1 and U2) may – and often will – occur nearly simultaneously due to the controller sending independent updates in parallel, or a distributed CP running. In such instances, our distributed verification protocol will always converge independently of the order of the updates (*e.g.*, U1, U2) similar to distributed link-state protocols. Depending on the order of the updates being processed, unnecessary verification steps may be executed before reaching the final stable state. To reduce route flaps, and increase the overall network stability, *Carbide* provides a damping mechanism.

C. Correctness Computation & Output

Given the generation and update of the LEC tables as described above, *CPCheck* can quickly verify a broad range of network correctness requirements based on the LEC table and correctness requirement specification.

Localization of Requirements. Different from centralized verification, distributed verification need to convert the global requirement into local requirements (*CPSpec*) at each device. This conversion achieves two goals: (1) *Carbide*'s *CPSpec* can reduce each property into a subset of regular expressions, allowing us to use generic regular expression parsers for local verification, and (2) *Carbide* can refine the quantity and specification of requirements by utilizing knowledge on which flows must pass through which device.

***CPCheck* output:** (HS_i^T, HS_i^F)**.** For a given control plane CP_i, the output from *CPCheck* is the set of packets HS_i^T for which CP_i provides correctness, as well as its complement,

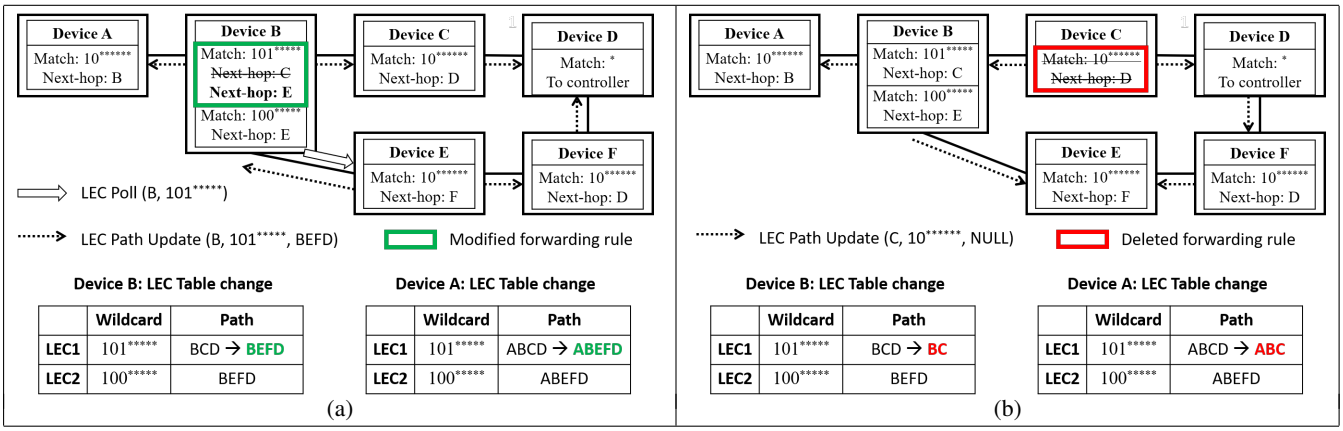


Fig. 3: *CPCheck* behavior in response to (a) a modified forwarding rule at B (update $U1$), and (b) a deleted forwarding rule at C (update $U2$).

HS_i^F. To calculate HS_i^T, *CPCheck* performs two steps at each device using locally available information: (1) for each property $prop_j$, calculate S_j^i, the set of packets for which CP_i satisfies the localization of $prop_j$, and (2) aggregate the S_j^i's into the desired result. To perform (1), *CPCheck* can simply take the union of all LECs whose paths (as given in the device's LEC table) successfully match the localization of $prop_j$. To perform (2), let $HS_i^F = \cup_j(DHS_j \setminus S_j^i)$, the set of all packets for which CP_i fails to satisfy the requirements, and let $HS_i^T = (HS_i^F)^c$. Such results $\{(HS_i^T, HS_i^F)\}, i = 1, 2, 3...$ for all CPS are taken as an input of the *CP Composer* to output a CP assignment table, which maps sets of packets to the best usable control plane.

CPCheck is capable of detecting network events such as forwarding rule update and link failure. When such an event is detected locally, an LEC update message is created, which is a tuple of the form (n, HS_{aff}, P). Here, n is the device where the event is detected, HS_{aff} is the entry which is affected by the change, and P is the new path to be taken by packets which have been following this LEC entry until now. Since each HS_{aff} corresponds to a certain property, all the routers that store and use this property need to be informed about the update. New routers that now participate in the updated path have to adopt this LEC entry into their corresponding tables, and the routers that do not participate need to remove it.

D. Link Sensitive Updating

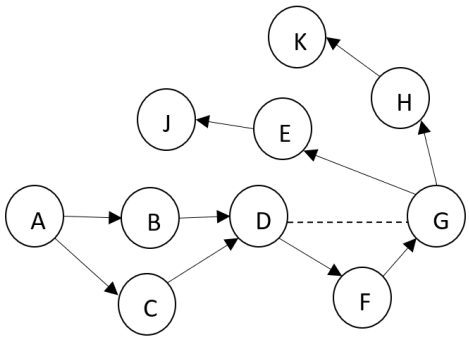
If a router senses that a packet it recently forwarded has been dropped or that a packet using a certain path is expected to be dropped in a near future (which is decided through mobility tracking discussed in Section IV-B), *CPCheck* initiates a protocol that allows *Carbide* to update the LEC table and immediately forward the packet through another, functional channel. In a wireless environment, link metric values are used to keep track of the quality of communication channel between any two routers. The metric itself may be calculated using measurements such as SNR (signal to noise ratio), transmission time, and link bandwidth [3], [7]. These values can be used to inform the routers that a link could be failing and it would be prudent to switch to an alternative route. The mechanism for forwarding the update message in a wireless environment is discussed in Section IV-A. In *Carbide*, every time the value of the link metric falls below a predefined threshold, we immediately trigger the process that updates equivalence class table. In this situation, since MANET protocols including OLSRv2 are expected to keep their FIB updated [6], [11], we can derive new equivalence class entry using the forwarding rule of such protocols. Since we know that there are first-hop neighbors that use the same equivalence class entry, the notification of failure should to be relayed to those routers quickly, along with the information on the updated route. In order to transmit the message of this link failure, to all the participant routers for this path, we introduce the idea of *Carbide* verification message, which is explained in Section IV-A. When a router is notified about the path update by the verification method, the router updates its LEC table to include the updated routes.

IV. Carbide in SDC Networks

Due to the dynamic and mobile nature of SDC networks, several complications may arise in the deployment of *Carbide*. First, broadcasting verification messages may significantly consume the bandwidth of the SDC network. Second, mobility may cause rapid changes in the network topology. Third, running multiple control planes and verification module affects energy consumption and resource utilization, which is a significant concern on mobile devices. To address these three challenges, we present a novel optimized messaging protocol, mobility tracking model, and the adaptive resource control scheme. We then introduce SDC extensions using *Carbide*'s verification module.

A. Optimized Flooding with Carbide Verification Message

Carbide verification messaging borrows some ideas from optimized broadcasting and multipath relay [6] employed by the OLSR protocol. This verification message will be broadcasted every time a router makes changes to a certain entry in the equivalence class table locally, and needs to convey the information about this change to all other participant routers who use this particular entry.While broadcasting can ensure that all the routers will get the messages, it will unnecessarily consume the bandwidth of the entire topology, even though the LEC update might only be warranted for a very small section of the topology. Rather than conveying the message to all the routers, selective broadcasting is employed such that only the routers that utilize this particular equivalence class entry (the one that just got updated) will participate. All other routers will immediately drop the received message.



LEC entries for A:		
	Match	Path
LEC1	*DstIP =H, DstPort =22*	*ABDFGH*
LEC2	*DstIP =H, DstPort! =22*	*ACDFGH*

Fig. 4: Arrangement of routers where a failure of link DG triggers an update process. This leads to each participants (A, B, D, F, G and H) learning about the new path, LEC2, through *Carbide* verification messaging.

If a router senses that the link between two routers is down, *Carbide* initiates a search for all the LEC entries that involve this particular link. For each such entry, *Carbide* will find an alternative route and once the route is determined, prepare a message containing the old LEC entry as well as the new one. Then this message is sent to all the one-hop neighbors. Only the neighbors that participate in old or new paths will update their LEC tables and broadcast the packets. The rest will drop the packet to avoid unnecessary congestion. This way, all the participant get the message and the bandwidth usage is minimized.

In Fig. 4, consider an LEC entry for forwarding a packet from A to H. These packets have to go through DG initially. However, the link could break and, in such cases, an alternative route (i.e., the entry ABDFGH) will have to be selected. This means updating LEC entry in all the participant routers, which includes the participants of the new and the old routes. Let us assume that D is the first participant to realize the link is broken. It is able to figure out using OLSR's FIB that an alternative route (DFG) is available. After D updates its LEC entry, it needs to inform all the participants that the updated route has to be taken from now on. We employ selective broadcasting method to inform all the participants about the update without unnecessarily consuming bandwidth of the entire topology. When D broadcasts this message, B and C will update their corresponding tables and re-broadcast the message. A will also update its LEC entry, but it will not forward the message as it knows that it is the source. Similarly, H will update its its LEC entry but will not forward the message to K as it is the destination. Knowing that it has no role to play for this LEC entry, E will immediately drop the packet. Hence, links such as EJ will not be disturbed by the *Carbide* verification message.

Jitters *Carbide* verification messaging employs Jitters to ensure that packets do not collide with one another during broadcast. For instance, both B and C might be broadcasting the message for A at the same time, leading to a collision. Since the updates are expected to take place at the same time, randomized time stamps for broadcasting minimizes collision between the packets carrying the same message.

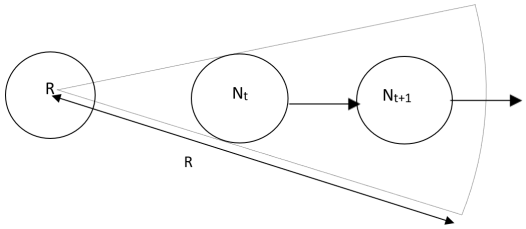


Fig. 5: Demonstration of receding 1-hop neighbor in a mobile environment. Router N is expected to stop being capable of communicating with R as it moves away and the link metric value between the routers is expected to fall below the required threshold m.

B. Mobility Tracking

To further improve the robustness of *Carbide*, we can employ predictive tracking of the mobile routers. By keeping track of deteriorations in the link metric values, we are able to update routing tables when it appears that a link is going to be broken. Some suggested methods for tracking routers in a wireless scenario include modified Kalman filtering [24], autoregressive models [25], and radio interferometry [15]. In vehicular ad-hoc networks, velocity has been used as a tracking mechanism for mobility tracking [20]. Aforementioned techniques can be specific to certain kinds of mobility environment, such as vehicular ad-hoc networks and unmanned aerial vehicles, and attempt to optimize communication in those specific environment. In our design, we take into consideration all possible situations and put emphasis on quick and non-intensive computation the within local *Carbide* itself. In this paper, we track the mobility of all one-hop neighbors of a router by observing the change in link metric values.

Consider a router R and its neighbor N as shown in Fig. 5. As the router N moves away from R, we can expect the link metric value to deteriorate, but the transfer of packets will continue right until the link metric value falls below the acceptable threshold, m. We can predict weather or not a packet is expected to be dropped based on current trajectory as long as the link metric value is deteriorating and m is known. So, if we observe the change in value at t and $t+1$ to be such that we can expect the value to fall below m at $t+2$, we preemptively initiate LEC entry update algorithm to avoid potential data loss. This way, when the router moves beyond the threshold, alternative routes will be used to deliver the packets that previously used RN as a link. This will also give us the time to let other participants of the particular LEC entry be informed about the update in routing policy.

In a wireless environment, *Carbide* attempts to optimize the delivery of packets by employing mobility-aware forwarding. For instance, based on the mobility parameters, special rules may be employed to ensure that routers with high mobility are provided with reliable communication channels. If a router is moving very fast and consequently sees its communication links change rapidly, *Carbide* will use a different routing mechanism to ensure that packets are still delivered.

C. Resource Control

Carbide consists of a multiple processes (e.g., *CPCheck* processes for each control plane) competing for different

resources (e.g., CPU, memory and I/O) within a device, and resource allocation among these processes is therefore critical. For example, if a *CPCheck* process of an control plane consumes all of a device's execution resources, other processes would starve and not be able to progress.

In particular, when an event happens (e.g., link failure), and multiple *CPCheck* processes are launched, and competing for resources, how should the resources be allocated? Intuitively, important processes should be assigned more resources. However, how should importance be defined in this specific context? In addition, uncertainty can further exacerbate the problem. For example, if more resources are assigned to a verification process which after running for a period of time returns False, would such resources have been better assigned to a different process?

To address this problem, we propose an *adaptive resource allocation* that takes into account both importance and uncertainty so that *Carbide* can operate properly, and efficiently.

Importance. We define importance of a process based on the following observations. First, a *CPCheck* process runs when there is an event causing a CP to change its FIB. Each *CPCheck* is associated with a CP, therefore, we consider the global preference order between the CPs specified by the administrator as a factor of importance. Second, the size of LEC (i.e., how many packets are belong to the LEC) for which *CPCheck* process is launched is proportional to the importance of that *CPCheck* process. Finally, the need for a *CPCheck* process to verify an LEC becomes less important if the LEC is verified by a verification process of another CP having higher preference order.

Uncertainty. A *CPCheck* process verification result can only be known after executing it. We therefore introduce the probability of a *CPCheck* that shows the likelihood of that process to verify its LEC. A *CPCheck* process with a higher probability is preferred.

Carbide uses the aforementioned importance and uncertainty factors as the utility functions of a *CPCheck* process; and we then simply allocate the resources to the different verification processes by maximizing the sum of their utilities subject to the total amount of resources available. We assign fixed amount of resources to the CP process and the CP composer. The CP composer controls all communication between the control planes and the data plane as shown in Fig. 1. Moreover, our evaluation shows that CP processes consume very limited amount of resources compared to *CPCheck*. In contrast, the *CPCheck* processes have shorter lifetimes, and are assigned dynamic amounts of resources determined by the adaptive resource allocation. A *CPCheck* process starts when a CP detects a change in a FIB. The *CPCheck* process goal consists in verifying a correctness requirement for a LEC. The adaptive resource allocation focuses on determining the resources allocated to each of the *CPCheck* processes, based on their utility function values.

D. Extension with Computation and Storage

Carbide's novel distributed verification protocol addresses not only network state verification but also mixed resource verification on SDC networks. *Carbide* allows coalition partners to enforce different requirements using *CPSpec* based language. For example, the controller in an SDC network can specify location-based preferences (*e.g.*, do not use a specific CP in a particular area to avoid network congestion) rather than simple ACL rules. Moreover, coalition partners can also specify resource-dependent properties such as shutting down CPU-intensive CPs after a critical battery level, especially for mobile devices.

	SDN	*Carbide*	OLSRv2
Average Downtime	27.34 ms	9.65 ms	42.53 ms

TABLE I: Average downtime for different CPs

V. Evaluations

In this section, we evaluate the effectiveness of *Carbide* in recovering from link failures.

Environment. We run the experiment in a virtualized environment on a Rocketfuel topology [2] (AS 1755). The routers runs as a separate Docker containers, with multiple CPs: an OLSRv2, and an SDN. Our evaluation shows the benefits of using multi CPs over an individual CP, and so, the evaluation could be replicated for a wider range of distributed protocols.

Methodology. We randomly select a pair of nodes and generate UDP traffic between them by running iperf [1].For the SDN CPs, we use precomputed paths. We then randomly select links to fail on one of the paths used by the UDP traffic. For the SDN CPs to recover from the failures, we implement a reactive approach. That is, when a failure happens, the device that detects it sends a control message to the controller to recalculate alternative forwarding paths and update the FIB of the affected devices accordingly [17]. For each run, we measure the downtime defined as the amount of time between the moment the destination stops receiving packets because of the failure to the moment the receiver starts receiving packets again. We obtain an average value from multiple runs.

Results. Our evaluation shows that while the average downtime of the SDN CP is 27.34 ms, the average downtime of OLSRv2 is 42.53 ms, and *Carbide* is even lower than both of them with 9.65 ms. *Carbide* can reduce the downtime of SDN by 65%, and that of OLSRv2 by 77%. The gain comes from two aspects. First, *Carbide* has the fastest recovery time compared to individual CPs. That is, in some runs, the SDN CP recovers faster, whereas in other runs, the OLSRv2 CP recovers faster; and in every case, *Carbide* switches to the one that recovers the fastest. Second, *Carbide* provides diversity between CPs. That is, if a failure does not affect all CPs, *Carbide* can switch to a non-failing CP without any throughput loss.

***Carbide* on EMANE.** We implement *Carbide* on EMANE (Extendable Mobile Ad-hoc Network Emulator) [18] to demonstrate that *Carbide* works reliably and efficiently when subjected to conditions prevalent in the tactical settings. We conduct the experiments on the NATO IST-124 Anglova scenario [19] with 24 nodes, where each node is deployed on a virtual machine (VM). Each VM having 2 cores with 4 GB of RAM runs EMANE, Quagga's Zebra daemon to communicate routing updates, OLSRv2, and *Carbide*'s novel distributed verification runtime. We also deploy a controller VM to run an SDN controller as well as the Anglova scenario. Our results show that *Carbide* can complete LEC calculations on a device less than 2.5 ms.

VI. Related Work

Recent work on network verification and control plane composition focuses on the following directions:

Centralized Verification. A number of methods have been proposed to verify network behaviors. HSA [13] provides a general framework to statically check network specifications

based on a snapshot of network state. NetPlumber [12], Veriflow [14], and Delta-net [10] can detect violations of network-wide invariants in the data plane of SDN networks in real-time. Minesweeper [4] and ARC [8] verify the configuration files of distributed control planes in an offline manner, by encoding the problem into logic formulas that can be checked for satisfiability by constructing a Binary Decision Diagram or calling an SAT/SMT solver. ATPG [26] and NetSight [9] verify and diagnose the run-time data plane by debugging the traces and histories of probing packets. This line of work relies on a centralized entity to collect network states or configuration files, making the system less resilient to device or link failures. *Carbide* provides greater robustness by utilizing a distributed verification algorithm while opportunistically utilizing centralized resources when available.

Hybrid SDN. Systems composed of layers of multiple control planes have also been proposed, such as [21], [22]. In these systems, in the event of the failure of a control plane, the composition allows for usage of a backup control plane to ensure reliability. Tilmans, et al. [21] designed an architecture, "IGP-as-a-Backup to SDN", in which each device contains a local software agent that runs a link-state IGP as a backup to traditional SDN routes. However, their model does not allow for any policy guarantees aside from traditional SDN, and also restricts the distributed protocol to only link-state protocols. Vissicchio, et al. [22] provides a new classification scheme of routing protocols based on their interactions with the Routing Information Base (RIB) and Forwarding Information Base (FIB) of network devices and provide sufficient conditions to prevent routing loops and black holes. However, their analysis does not provide guidance or mechanisms on guaranteeing other desirable properties such as waypoints or route symmetry. In contrast, *Carbide* utilizes a unified distributed verification approach associated with each control plane to support a broad range of correctness properties for each packet in the network.

ACKNOWLEDGMENT

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

REFERENCES

[1] Iperf, a tool for active measurements of the maximum achievable bandwidth on ip networks. https://iperf.fr/.

[2] Rocketfuel: An isp topology mapping engine. http://www.cs.washington.edu/research/networking/rocketfuel.

[3] R. Aguero, J. A. Galache, and L. Munoz. Using snr to improve multihop routing. In *VTC Spring 2009 - IEEE 69th Vehicular Technology Conference*, pages 1–5, April 2009.

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168. ACM, 2017.

[5] T. Clausen, C. Dearlove, P. Jacquet, and U Herberg. The optimized link state routing protocol version 2, 2014.

[6] T. Clausen and P. Jacquet. Optimized link state routing protocol (olsr), 2003.

[7] Saumitra M. Das, Himabindu Pucha, Konstantina Papagiannaki, and Y. Charlie Hu. Studying wireless routing link metric dynamics. In *Proceedings of the 7th ACM SIGCOMM Internet Measurement Conference, IMC 2007, San Diego, California, USA, October 24-26, 2007*, pages 327–332, 2007.

[8] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. Automatically repairing network control planes using an abstract representation. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 359–373. ACM, 2017.

[9] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, volume 14, pages 71–85, 2014.

[10] Alex Horn, Ali Kheradmand, and Mukul R Prasad. Delta-net: Real-time network verification using atoms. In *NSDI*, pages 735–749, 2017.

[11] Y. Huang, S. N. Bhatti, and D. Parker. Tuning olsr. In *2006 IEEE 17th International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1–5, Sep. 2006.

[12] Peyman Kazemian, Michael Chan, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *NSDI*, pages 99–111, 2013.

[13] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, volume 12, pages 113–126, 2012.

[14] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 15–27, 2013.

[15] Branislav Kusy, Janos Sallai, Gyorgy Balogh, Akos Ledeczi, Vladimir Protopopescu, Johnny Tolliver, Frank DeNap, and Morey Parang. Radio interferometric tracking of mobile wireless nodes. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, MobiSys '07, pages 139–151, New York, NY, USA, 2007. ACM.

[16] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, pages 234–244, New York, NY, USA, 1994. ACM.

[17] Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester. A demonstration of fast failure recovery in software defined networking. In *International Conference on Testbeds and Research Infrastructures*, pages 411–414. Springer, 2012.

[18] Eric Schreiber Steven M. Galgano, Kaushik B. Patel. EMANE user manual 0.8.1. https://downloads.pf.itd.nrl.navy.mil/docs/emane/emane.pdf, 2013.

[19] N. Suri, J. Nilsson, A. Hansson, U. Sterner, K. Marcus, L. Misirliolu, M. Hauge, M. Peuhkuri, B. Buchin, R. in't Velt, and M. Breedy. The angloval tactical military scenario and experimentation environment. In *2018 International Conference on Military Communications and Information Systems (ICMCIS)*, pages 1–8, May 2018.

[20] T. Taleb, E. Sakhaee, A. Jamalipour, K. Hashimoto, N. Kato, and Y. Nemoto. A stable routing protocol to support its services in vanet networks. *IEEE Transactions on Vehicular Technology*, 56(6):3337–3347, Nov 2007.

[21] Olivier Tilmans and Stefano Vissicchio. Igp-as-a-backup for robust sdn networks. In *Network and Service Management (CNSM), 2014 10th International Conference on*, pages 127–135. IEEE, 2014.

[22] Stefano Vissicchio, Luca Cittadini, Olivier Bonaventure, Geoffrey G Xie, and Laurent Vanbever. On the co-existence of distributed and centralized routing control-planes. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 469–477. IEEE, 2015.

[23] Qiao Xiang, Franck Le, Yeon-Sup Lim, Vinod Mishra, Christopher Williams, Y Richard Yang, and Hongwei Zhang. Opensdc: A novel, generic datapath for software defined coalitions. pages 1–6, 10 2018.

[24] Z. R. Zaidi and B. L. Mark. A mobility tracking model for wireless ad hoc networks. In *2003 IEEE Wireless Communications and Networking, 2003. WCNC 2003.*, volume 3, pages 1790–1795 vol.3, March 2003.

[25] Z. R. Zaidi and B. L. Mark. Mobility tracking based on autoregressive models. *IEEE Transactions on Mobile Computing*, 10(1):32–43, Jan 2011.

[26] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 241–252. ACM, 2012.