Reclaiming the Brain: Useful OpenFlow Functions in the Data Plane

Liron Schiff*[1], Michael Borokhovich[2], Stefan Schmid†[3]
[1] Tel Aviv University, Israel; [2] University of Texas at Austin, United States; [3] TU Berlin & T-Labs, Germany

ABSTRACT

Software-defined networks (SDNs) have the potential to radically simplify the network management by providing a programmatic interface to a logically centralized controller. However, outsourcing the management to the software controller comes at a price, and good tradeoffs have to be found between the benefits of a fine-grained control and its costs.

In this paper, we show that OpenFlow, the predominant SDN protocol, allows to implement powerful functions "in the south", i.e., in the data plane. Our approach, called ***SmartSouth***, can be used to reduce interactions with the control plane as well as to make the network more robust. Moreover, while rendering the data plane "smarter", ***SmartSouth*** only relies on the standard OpenFlow match-action paradigm; thus, the data plane functions remain formally verifiable—a key benefit of SDN. To demonstrate the potential of ***SmartSouth***, we discuss four basic applications: (1) topology snapshot, (2) anycast, (3) blackhole- and (4) critical node detection.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms

Algorithms, Performance

Keywords

In-band Mechanisms; SDN; Troubleshooting

*Supported by the European Research Council (ERC) Starting Grant no. 259085 and by the Israel Science Foundation Grant no. 1386/11.

†Supported in part by the FP7 UNIFY EU project.

HotNets-XIII, October 27-28, 2014, Los Angeles, CA, USA.

http://dx.doi.org/10.1145/2670518.2673873 .

1. INTRODUCTION

Software-defined networks (SDNs) expose an API which facilitates the programmatic network management from a logically centralized software controller. This API also allows to split the network into a "dumb" *data plane* whose main purpose is to process and forward packets according to the rules installed by the network operating system, i.e., the "smart" controllers in the *control plane*.

While software-defined networking is a promising paradigm which can simplify and automate the network management, the separation of the control plane from the data plane comes with limitations and overheads. For instance, a fine-grained and reactive control of the data plane can induce high communication and computation cost in the control plane, constraining scalability [5, 7, 9], and introducing undesirable latencies [10, 16]. Indeed, in an SDN, the controller typically only has an imperfect visibility of the data plane and its events, which can render troubleshooting more challenging [4, 6]; sometimes, data plane elements may even lose connectivity to the control plane entirely [13].

Accordingly, there is an ongoing debate on the granularity of control and visibility which should be supported in the control plane as well as the functionality which should be kept in the data plane. [5] For example, newer versions of OpenFlow introduce local fast failover mechanisms which allow the data plane to quickly react to failures, without the interaction of the control plane; the control plane can improve the resulting flow allocation and network configuration in a second stage. [10, 12, 15]

Interestingly, while the question of which functionality ***should*** be provided in the data plane (and at what granularity) is discussed intensively today (e.g., [5, 7, 10, 17]), surprisingly little is known about the fundamental question of what functionality ***can*** be implemented using the current OpenFlow protocol.

Our Contribution. While the study and design of in-band mechanisms obviously have a long tradition [18, 19], to the best of our knowledge, this paper is the first to point out the potentially rich functionality of an *existing OpenFlow data plane*. By additionally leveraging

the OpenFlow fast failover mechanism, the data plane functions can also be made robust to failures. Thus, such functionality may not only be attractive to reduce the load on the control plane, but may also be used to provide critical information to troubleshooting applications in a resilient manner.

The approach introduced in this paper relies on a simple template called ***SmartSouth***. While ***SmartSouth*** renders the data plane, the "south", smarter, it does not assume any new hardware or protocol features (beyond OpenFlow 1.3). This is interesting, as it allows to keep the data plane and its forwarding state formally verifiable, that is: the key benefits and philosophy of SDN are preserved.

To demonstrate the possible applications of our mechanism, we present four concrete data plane functions.

1. *Snapshot:* The snapshot function allows to collect a view of the current network topology. The snapshot service is fault-tolerant in the sense that it does not assume any knowledge about the underlying network, nor direct connectivity to controllers (unlike, e.g. *TopologyService* [1]).

2. *Anycast:* The anycast function supports the specification of multiple legal destinations (e.g., different nodes providing the same functionality). If any of these destinations can be reached, the packet will be delivered. The anycast function can also be extended to support priorities over destinations (e.g., first only the destination with the highest priority is tried), or to specify entire service chains [11, 14] (e.g., a set of middleboxes which need to be traversed). The anycast service can for example be useful to find an alternative in-band path to a given controller after link failures, or to find a path to an alternative (and close) controller in case the control plane is distributed [7, 17].

3. *Blackhole Identification: Blackholes*, also known as "silent failures", especially occur in evolving networks. [8] The blackhole identification function allows us to actively discover blackholes, and to raise an alarm whenever end-to-end connectivity is disrupted, regardless of the cause. As we will show, we cannot only detect elements that drop *every* packet, but we can also monitor packet-loss at network links. As we will see, the blackhole identification function is an example of a service which needs some limited interaction between the "smart" data plane and the controller.

4. *Critical Node Detection:* The critical node detection function determines whether a certain node in the network is critical for the connectivity of the underlying physical network. (Or could, e.g., be removed or turned off for maintenance or energy conservation purposes.) While a critical node could also be detected using the snapshot function, computing the entire snapshot is costly and, as we will see, not needed.

Background and Model. The SDN network operating system consists of a control software running on a set of servers. These *controllers* receive information and statistics from switches, and depending on this information as well as the policies they seek to implement, issue instructions to the switches.

OpenFlow, the standard SDN protocol today, follows a *match-action paradigm*: The controllers install rules on the switches which consist of a match and an action part; the packets (i.e., flows) matching a rule are subject to the corresponding action. That is, each switch stores a set of tables which are managed by the controllers, and each table consists of a set of flow entries which specify expressions that need to be matched against the packet headers, as well as actions that are applied to the packet when a given expression is satisfied. Possible actions include dropping the packet, sending it to a given egress port, or modifying it, e.g., adding a *tag*. Indeed, the mechanisms presented in this paper will make use of the tagging mechanism to equip packet instances with meta-information.

Concretely, each switch has multiple *flow tables* and a *group table*. Each flow table in the switch contains a set of *flow entries*; each flow entry consists of match fields, counters, and an ordered list of *action buckets*. Each action bucket contains a set of *actions* to execute, and provides the ability to define multiple forwarding behaviors. The group table consists of multiple groups, where different groups can have different types. For instance, the type `ALL` can be used to execute all buckets in the group; and the *fast failover* type `FF` can be used to render data plane mechanisms robust: a bucket in this table is associated with a parameter that determines whether the bucket is live; a switch will always forward traffic to the first live bucket. To determine liveness, the programmer specifies an output port.

2. THE SMARTSOUTH MECHANISM

The data plane functions presented in this paper are realized in two stages. In the first stage, the "offline stage", the OpenFlow tables are installed at the switches, implementing the desired functionality. In the second stage, the "runtime stage", the distributed data plane function is executed: the execution is triggered by the injection of certain requests (henceforth sometimes called the *trigger packet*). Depending on the application, the second stage does not require any or only very limited controller interactions, i.e., no modifications to the data plane configuration are made.

All data plane functions presented in this paper are based on a simple template, henceforth called ***SmartSouth***. ***SmartSouth*** is essentially an in-band

graph traversal mechanism implemented in the standard match-action paradigm; it is based on OpenFlow tables (at most version 1.3). By using fast failover techniques (implemented with the FF tables), the template becomes robust to link failures. (However, in the following, we will assume that *during* the execution of ***SmartSouth***, no more failures will occur. This limitation can be overcome by using e.g. mechanisms presented in [3].)

SmartSouth implements the first stage of the function realization: it installs a set of rules and tables in the data plane. Essentially, ***SmartSouth*** constructs a spanning tree, i.e., when the function execution is triggered in the second stage, a trigger packet will traverse the network according to a depth first search (DFS) order.

The ***SmartSouth*** template is summarized in Algorithm 1. It is parametrized with different service types (snapshot, anycast, blackhole detection, critical node detection), and we present the service specific functions in detail later. Generally, the template first determines whether the current switch is the first to execute the requested function (in which case it becomes the *DFS root*), or whether the function has been started by some other node. During the function execution, a single trigger packet traverses the network, subject to the installed match and action rules. For each node i, we reserve a certain number of bits in the packet header, the *tag*, where the node can store the port of its parent ($pkt.v_i.par$), as well as the port of the neighbor it is currently visiting ($pkt.v_i.cur$). Additionally, the packet header includes a global tag field $pkt.start$ which indicates whether the service has already started. As we will see, more tag fields will be introduced by the specific service. We denote the total number of ports at node i by Δ_i, and assume that all the tag fields are initialized to 0. When a node i sees a packet for the first time, it sets $pkt.v_i.par$ and starts to traverse its own neighbors by first trying output port 1 (and setting $pkt.v_i.cur \leftarrow 1$). In case that the port failed (i.e., is not *live*), the next live port is selected. Once all the neighbors were traversed, the node returns the packet to its parent.

Note that since the underlying network topology is typically not a tree, a node may receive a packet from an "unexpected" port (i.e., not from $pkt.v_i.cur$). In this case, the node will just return the packet to the port from which it was received. Once the root node finishes traversing all its neighbors (port *out* will be 0 only at the root node when it wants to send the packet to its parent, which is 0), ***SmartSouth*** terminates.

In addition, each data plane service that we will describe in the following, implements some service-specific functions in the template. These functions determine the behavior of the service upon packet reception at the different stages of ***SmartSouth***. The first function is $First\_visit()$: it specifies the actions to be executed when a node (other than the root) sees a service packet for the first time. The function $Visit\_not\_from\_cur()$ describes the behavior of a node i once it receives an "unexpected" packet, i.e., a packet arriving from a port different from $pkt.v_i.cur$. The $Send\_next\_neighbor()$ function is called when a node i has received a packet from the "expected" port (i.e., $pkt.v_i.cur$), and forwards it to the next neighbor, and the $Send\_parent()$ function is called when a packet is returned to the parent. The $Finish()$ function is called only by the root node (since only the root has an *out* port 0 once it tries to forward the packet to the parent).

Algorithm 1 Algorithm ***SmartSouth*** – Template

Input: current node: v_i, input port: in, packet global params: $pkt.start$, packet tag array: $\{pkt.v_j\}_{j\in[n]}$
Output: output port: out

```
 1: if pkt.start = 0 then
 2:     pkt.start ← 1
 3:     out ← 1
 4: else
 5:     if pkt.v_i.cur = 0 then
 6:         pkt.v_i.par ← in; out ← 1; First_visit()
 7:     else if in = pkt.v_i.cur then
 8:         out ← pkt.v_i.cur + 1; Visit_from_cur()
 9:     else
10:         out ← in; Visit_not_from_cur()
11:         goto 26
12:     if out = Δ_i + 1 then
13:         out ← pkt.v_i.par
14:         goto 22

15: while out failed or out = pkt.v_i.par do
16:     out ← out + 1
17:     if out = Δ_i + 1 then
18:         out ← pkt.v_i.par
19:         goto 22

20: Send_next_neighbor()
21: goto 23
22: Send_parent()
23: pkt.v_i.cur ← out
24: if out = 0 then
25:     Finish()
26: return out
```

3. CASE STUDIES

To demonstrate the power of ***SmartSouth*** data plane functions, we present four case studies. See Table 1 for details of these implementations.

3.1 Snapshots

The computation of topological snapshots is a fundamental task in any distributed system. For example, a snapshot can be useful for network troubleshooting applications. We will study the following scenario: upon request, a list of all currently active nodes and their active links is compiled and sent back to the requester. For ease of presentation, we will first assume that there is sufficient space in the packet to include a full snapshot;

we will later discuss how the OpenFlow data plane can also split the snapshot across multiple packets if needed.

We extend ***SmartSouth*** as follows: 1) record node ids and in-ports each time a node is visited; 2) record out-ports before the node is left. This can be achieved by writing to the reserved space in the packet header (possibly using a counter to find the next available location) or by pushing labels, each time it visits a node.

Our approach tracks edges that are not describing parent-child relationships in the tree *twice*. To save packet header space we distinguish between the two visits. We split the $Visit\_not\_from\_cur()$ function into two sub-cases depending on whether $in < pkt.v_i.cur$ or not. If $in < pkt.v_i.cur$ then node i has already used the incoming port itself, and hence the packet is now received from an ancestor which is not the parent of node i. We handle this case by deleting the tracking made by the sender ($pop()$) instead of applying the default behavior of adding more tracking information. In the second sub-case, we apply the default behavior. Notice also that if node i has already finished traversing its neighbors and sent the packet to its parent, its $pkt.v_i.cur$ is set to $pkt.v_i.par$, which may affect the $in < pkt.v_i.cur$ comparison. Thus, in the case where $pkt.v_i.cur = pkt.v_i.par$, we act as if $in < pkt.v_i.cur$. Note that comparing two fields is not directly supported by OpenFlow but it can be implemented with a dedicated flow tables [2].

Remarks. If the snapshot of a large network does not fit into a single packet, data plane mechanisms can be implemented to split a packet into multiple smaller ones. All we have to do is to track the amount of data gathered so far (e.g. using special counter) and, when needed, we send the packet to the controller.

3.2 Anycast

Anycast provides the possibility to send a message to *any* node in a specified group of potential receivers $\mathcal{R}$. Such a function could for example be used to ensure that a packet always passes through a node providing certain functionality (e.g., deep packet inspection); another application is a distributed control plane [7, 17] where a packet must reach a close controller. Anycasts can easily be chained, in the sense that sequences of middleboxes can be specified which need to be traversed. [14]

The anycast can be realized by introducing a field named *gid* in the anycast message that stores the *id* of the anycast group. Concretely, we add a simple test at the beginning of the ***SmartSouth*** template. In every node i, this test tries to match all ids of groups that i belongs to, and a successful match triggers the forwarding of the packet to a predefined ("self") port; otherwise, the remainder of the traversal code is executed, allowing the packet to reach all available nodes until a relevant receiver is found. This can easily be translated to flow table rules: for each group id we have an entry in the first flow table that matches the packet *gid* field with the relevant group id.

The anycast function can also be extended to support *priorities* $p_{i,gid}$ over destinations $v_i \in \mathcal{R}$ where $\mathcal{R}$ has id *gid*: i.e., destinations are tried in decreasing priority order. We name this extension *priocast*. For instance, priocast could be useful to find an alternative in-band path to the controller, if the management port of the controller cannot be reached.

We implement priocast by making two phase traversals: in the first phase, the optimal receiver is computed by having each possible receiver updating the currently best receiver id in the packet. In the second phase, the root issues a second traversal in order to direct the packet to the optimal receiver found in the previous phase. The priocast service can be realized by modifying ***SmartSouth*** as follows (see Table 1). We introduce two new packet fields named *opt_id* and *opt_val* which are used to store the id of the best receiver found so far, as well as its priority. Moreover, we extend the *start* field to be ternary, holding the current phase, where 0 represents, as before, an uninitialized search. We use these fields in the following way: if *start* is 1, we match at any node i, the *gid* packet field to all group ids that i belongs to (represented by $GIDS_i$ in the pseudo code); in the case of a successful match to id, *gid* of group $\mathcal{R}$, we check whether i has a higher priority than the best node found so far, i.e., $p_{i,gid} > opt\_val$. In case the inequality holds, we update *opt_id* and *opt_val* with values i and $p_{i,gid}$ respectively. In any case, we continue the traversal to allow consideration of all nodes. The $Finish()$ function in the $start = 1$ case is used to set *start* to 2, and begin a new traversal by setting the next out port to the first one used in the first traversal.

Non-root nodes detect the phase switch by matching the in-port to its parent field (which can happen only when an additional traversal starts). In this case *out* is initialized to 1 and the processing is similar to the first visit case. Moreover, in the second phase each node i checks whether it is the optimal receiver, i.e. $i = opt\_id$, and if this is the case, it outputs the packet to (port) "self". If the test fails, the node continues the traversal, eventually reaching the optimal receiver.

3.3 Blackhole Detection

Modern multi-layer networks face the challenge of "silent failures", failures which are not observed directly. Such blackholes may arise due to physical failure, configuration errors, or general unsupervised carrier network errors. [8] In the following, we show how to determine whether there exists an edge (or similarly: port) that loses all packets. Concretely, we present two different solutions with a different level of controller involvement and message complexity.

The first solution executes multiple DFS edge traver-

	service			
	priocast	*snapshot*	*blackhole*	*critical*

First_visit()	**if** $pkt.gid \in GIDS_i$ **then**
	  if $pkt.opt\_val < p_{i,gid}$ **then**
	  $pkt.opt\_val \leftarrow p_{i,gid}$
	  $pkt.opt\_id \leftarrow i$
		$push(\{i, in\})$	**if** $pkt.repeat = 3$ **then**
			  $pkt.repeat \leftarrow 2$
			  $out \leftarrow \Delta_i$
				-

Visit_from_cur()	**if** $in = pkt.v_i.par$ **then**
	  if $i = pkt.opt\_id$ **then**
	  $out \leftarrow SELF$
	  else
	  $out \leftarrow 1$
		-	**if** $in = pkt.v_i.par$ **then**
			  $pkt.repeat \leftarrow 3$
			  $out \leftarrow 1$
			else if $pkt.repeat = 2$ **then**
			  $FetchAndInc(C_{in})$
			  $pkt.repeat \leftarrow 1$
			  $out \leftarrow in$
			else if $pkt.repeat = 1$ **then**
			  $FetchAndInc(C_{in})$
			  $pkt.repeat \leftarrow 3$
				-

Visit_not_from_cur()	-	**if** $in < pkt.v_i.cur$ **or** $pkt.v_i.cur = pkt.v_i.par$ **then**
		  $pop(\{x\})$
		else
		  $push(\{i, in\}, \{out\})$
			if $pkt.repeat = 3$ **then**
			  $pkt.repeat \leftarrow 1$
			  $out \leftarrow in$
				if $pkt.v_i.par = 0$ **and** $pkt.v_i.cur \neq pkt.firstPort$ **and** $pkt.toParent = 1$ **then**
				  $SendController(crit = true)$
				$pkt.toParent \leftarrow 0$

Send_next_neighbor()	**if** $pkt.v_i.par = 0$ **and** $pkt.v_i.cur = 0$ **then**
	  $pkt.firstPort \leftarrow out$
		$push(\{out\})$	**if** $pkt.repeat = 3$ **then**
			  $Fetch\&Inc(C_{out})$
			else if $pkt.repeat = 0$ **then**
			  if $Fetch\&Inc(C_{out}) = 1$ **then**
			  $pkt.v_i.cur \leftarrow out$
			  $SendController()$
				if $pkt.v_i.par = 0$ **and** $pkt.v_i.cur = 0$ **then**
				  $pkt.firstPort \leftarrow out$
				$pkt.toParent \leftarrow 0$

Send_parent()	-	-	-	$pkt.toParent \leftarrow 1$

Finish()	**if** $start = 1$ **then**
	  $start \leftarrow 2$
	  $out \leftarrow pkt.firstPort$
	  $pkt.v_i.cur \leftarrow pkt.firstPort$
	else
	  Drop
		$SendController(pkt)$	$SendController$ ("No Blackhole")	$SendController(crit = false)$

Table 1: *SmartSouth* service functions.

sals, each with a different Time-To-Live (TTL) value. The goal of these traversals is to find, in a binary-search fashion, the exact point along the DFS tree where the packet is lost. The code of the original DFS in ***SmartSouth*** is hardly changed, we only add functionality for TTL inspection and increment. Upon each visit, a node checks whether the TTL value is 0, and if so, records the edge it was about to forward to (if the TTL had not been 0); this information is forwarded to the controller. For each request, the application waits whether the request returns, and if yes, checks whether there exists a blackhole farther in the DFS.

As a second approach, we present a more complicated yet more effective solution, which only requires two out-band packets. At the heart of this solution lies a technique we call *smart counters*.

Smart Counter. A smart counter implements multiple (small) counters that are stored in the switch and support fetch-and-increment operations while processing different packets. The smart counter can be read and updated while packets are processed in the OpenFlow pipeline. The implementation is based on defining a group with *round-robin bucket selection* policy (an optional feature of OpenFlow 1.3). The number of buckets equals the number of counter values, i.e., k buckets are required for a counter with values $\{0, ..., k-1\}$. For each bucket, we define an action that writes its sequence (the position of the bucket in the group) to some packet header field (e.g., a temporary field), allowing it to be matched and used by the flow tables. This allows to have increasing bucket ids for a packet applied to the group, and to fetch nd increase the counter; an overflowing counter returns to 0.

Probing Links with Smart Counters. To count the number of messages that traverse a link, our blackhole algorithm proactively installs one smart counter per switch port (represented as C_x, where x is the port number). Given such counters, our algorithm uses one network traversal that goes back and forth once on every new link (using a special packet field named *repeat*): Accordingly, the counter of every non-blackhole link is increased to a value greater than 1, while the counters of blackhole links (port) have value one. Subsequently, our algorithm initiates a second traversal which, upon

detecting the counter-1 link, sends its description to the controller. The *repeat* field is also used to differentiate the packets of the two traversals ($repeat = 0$ in the second traversal). The controller sends the two packets with a time difference of twice the maximum delay, and then waits for the blackhole report.

Detecting Packet-Loss with Smart Counters. Our mechanism cannot only be used to handle links that drop *every* packet (and more importantly, links that may drop our in-band blackhole detection packets), but also to monitor *packet loss*. The main idea is to use two (additional) smart counters per port, one counter for outgoing packets and the other one for incoming packets. Every time a packet is sent through a port, the outgoing counter of a link is increased; and analogously for in-ports when a packet is received. When our blackhole detection packet traverses the network it triggers the comparison of outgoing and incoming counters on opposites sides of the links; if the counters differ, a packet-loss is reported. Note that counters may overflow (and be reset to 0), and accordingly, a packet may be lost (a false negative); as a possible solution, we suggest to increase and compare a few smart counters, with unique and prime sizes.

3.4 Critical Nodes

In this case study, we want to determine whether a given switch v_i is *critical*: Will the removal of v_i lead to a network partition? We implement the data plane function to realize this application as follows: the controller asks a node to check its own *criticality*, by sending it a special packet. Upon reception of the packet, the node starts the traversal from port 1 (or, if port 1 is not live, the first consequent working port). This first port is stored in the packet in the $pkt.firstPort$ field. If the node is not critical, the packet sent from $pkt.firstPort$ will traverse all the nodes in the network, before it returns back via this port. Thus, if i is not critical, then no other node except of the neighbor at port $pkt.firstPort$ will choose i as the parent in further traversal process.

Now, in order to be able to verify that i is not selected as a parent for its neighbor, we add a bit field $pkt.toParent$ in the tag of the packet which will be set to 1 when a node sends a packet to its parent. So, once the packet is back from the $pkt.firstPort$ port, the node i advances $pkt.v_i.cur$ to the next working port, sends the packet via this port, and continues the ***SmartSouth*** traversal. From this point on, the node i inspects the $pkt.toParent$ field of all the incoming packets. Once it sees a packet with $pkt.toParent = 1$ it immediately decides that it is a critical node, and informs the controller. If the ***SmartSouth*** traversal is finished and no additional node selected i as the parent, then i informs the controller that it is not a critical node.

Remark on Complexity. We conclude this section with a discussion of the overhead of the different ***SmartSouth*** services. Clearly, the anycast variants do not require any out-of-band messages, and only a constant number of messages (of constant size) are needed for the blackhole and critical node services. The complexity of the snapshot is simply given by the size of the network that needs to be collected; as discussed, the snapshot may also be split into multiple messages. The number of in-band messages of ***SmartSouth*** is in the order of the network size as well ($|E|$ links are visited). Only the messages of snapshot may become large. Note that all out-of-band messages can be sent in-band to any server connected to the first node of the traversal, thereby allowing complete in-band monitoring.

Table 2 summarized the complexities. In general, using switches like our NoviKit 250 switch (32MB flow table space and full support for extended match fields) and if the size of the data section of packets is limited to 0.5KB, we believe that our algorithms scale up to a few hundred nodes.

Service	Complexity	
	out-band #msgs×size	in-band #msgs×size
Snapshot	$1 \times O(1) + 1 \times O(\|E\|)$	$(4\|E\| - 2n) \times O(\|E\|)$
Anycast	0	$(4\|E\| - 2n) \times \|data\|$
Priocast	0	$(8\|E\| - 4n) \times \|data\|$
Blackhole 1	$2 \log \|E\| \times O(1)$	$(8\|E\| - 4n) \times O(1)$
Blackhole 2	$3 \times O(1)$	$4\|E\| \times O(1)$
Critical	$2 \times O(1)$	$(4\|E\| - 2n) \times O(1)$

Table 2: Overview of the complexities of the different *SmartSouth* services. The message size does not include the DFS part, which adds another $O(n \log n)$ bits, where n is the network size.

4. CONCLUSION

We understand our work as a first step, and believe that our results can nourish the debate on how to partition the functionality between the "dumb data plane" and the "smart control plane". While we kept our case studies simple and "inspirational" on purpose, our techniques can be extended to implement many other functions. For example, the smart counter concept introduced in this paper may also be used to infer network loads. Thus, we believe that our paper opens a rich field for future research.

Remark. Within the UNIFY project[1] we will investigate how ***SmartSouth*** could provide additional robustness for monitoring dynamically instantiated service chains.

[1]See http://www.fp7-unify.eu/.

5. REFERENCES

[1] `http://www.openflowhub.org/display/floodlightcontroller/TopologyService`. *openflowhub.org*, 2012.

[2] Y. Afek, A. Bremler-Barr, and L. Schiff. Cross-entrance consistent range classifier with openflow. In *Proc. Research Track of the Open Networking Summit (ONS)*, 2014.

[3] M. Borokhovich, L. Schiff, and S. Schmid. Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms. In *Proc. ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014.

[4] Colin Scott et al. Troubleshooting sdn control software with minimal causal sequences. In *Proc. ACM SIGCOMM*, 2014.

[5] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proc. ACM SIGCOMM*, pages 254–265, 2011.

[6] N. Handigol, B. Heller, V. Jeyakumar, D. Mazieres, and N. McKeown. Where is the debugger for my software-defined network? In *Proc. 1st Workshop on Hot Topics in Software Defined Networks (HotSDN)*, pages 55–60, 2012.

[7] S. Hassas Yeganeh and Y. Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proc. 1st Workshop on Hot Topics in Software Defined Networks (HotSDN)*, pages 19–24, 2012.

[8] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Detection and localization of network black holes. In *Proc. IEEE INFOCOM*, 2007.

[9] T. Koponen et al. Onix: A Distributed Control Platform for Large-scale Production Networks. In *USNIX OSDI*, 2010.

[10] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker. Ensuring connectivity via data plane mechanisms. In *Proce. 10th USENIX NSDI*, pages 113–126, 2013.

[11] P. Skoldstrom et al. Towards unified programmability of cloud and carrier infrastructure. In *Proc. European Workshop on Software Defined Networking (EWSDN)*, 2014.

[12] P. Pan, G. Swallow, and A. Atlas. Fast reroute extensions to RSVP-TE for LSP tunnels. In *RFC 4090*, 2005.

[13] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker. Cap for networks. Proc. HotSDN, 2013.

[14] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using sdn. In *Proc. ACM SIGCOMM*, pages 27–38, 2013.

[15] M. Reitblatt, M. Canini, A. Guha, and N. Foster. Fattire: Declarative fault tolerance for software-defined networks. In *Proc. HotSDN*, pages 109–114, 2013.

[16] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang. Leveraging zipf's law for traffic offloading. *SIGCOMM Comput. Commun. Rev.*, 42(1):16–22, Jan. 2012.

[17] S. Schmid and J. Suomela. Exploiting locality in distributed sdn control. In *Proc. SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2013.

[18] C.-C. Tu, P.-W. Wang, and T.-c. Chiueh. In-band control for an ethernet-based software-defined network. In *Proc. SYSTOR*, 2014.

[19] S. Zhang, L. Vanbever, and S. Malik. In-Band Update for Network Routing Policy Migration. In *Proc. IEEE ICNP*, 2014.