Answering Why-Not Queries in Software-Defined Networks with Negative Provenance

Yang Wu
University of Pennsylvania

Andreas Haeberlen
University of Pennsylvania

Wenchao Zhou
Georgetown University

Boon Thau Loo
University of Pennsylvania

ABSTRACT

When debugging an SDN, it is sometimes necessary to explain the *absence* of an event: why a certain rule was *not* installed, or why a certain packet did *not* arrive. Existing SDN debuggers offer some support for explaining the *presence* of events, usually by providing the equivalent of a "backtrace" in conventional debuggers, but they are not very good at answering "Why not?" questions: there is simply no starting point for a possible backtrace.

In this paper, we show that the concept of *negative provenance* can be used to explain the absence of events in SDNs. Negative provenance relies on counterfactual reasoning to identify the conditions under which the missing event *could have* occurred. We outline a simple technique that can track negative provenance in SDNs, and we present a case study to illustrate how our technique can be used to answer concrete "Why not?" questions. Using our approach, it should be possible to build SDN debuggers that can explain both the presence and the absence of events.

Categories and Subject Descriptors

C.2.3 [**Computer Systems Organization**]: Computer-Communication Networks—*Network Operations*

General Terms

Algorithms, Design, Reliability

Keywords

Software-defined Networks, Debugging, Negative Provenance

1. INTRODUCTION

Finding problems in complex networks has always been challenging, as the substantial literature on network debugging and root-cause analysis tools [4, 7, 10, 11, 13] can attest. However, the advent of software-defined networking (SDN) has added a new dimension to the problem: networks can now be controlled by programs, and, like all other programs, these programs can have bugs. Finding such bugs can be difficult because, in a complex network of routers, switches, and middleboxes, they can manifest in subtle ways that have no obvious connection with the root cause. It would be useful to have a "network debugger" that can assist the network operator with this task, but existing techniques tend to be protocol-specific and cannot necessarily be applied to SDNs with arbitrary control programs. Hence, as others [7, 17] have observed, a more powerful debugger is needed.

Existing solutions, such as `ndb` [7] or SNP [20], approach this problem by offering a kind of "backtrace", analogous to a stack trace in a conventional debugger, that can link an observed effect of a bug to its root causes. For instance, suppose the administrator notices that a server is receiving requests that should have been handled by another server. The administrator can then trace the requests to the last-hop switch, where she might find a faulty rule; she can trace the faulty rule to a statement in the controller program that was triggered by a certain condition; she can trace the condition to a packet from another switch; and she can continue to recursively explain each effect by its direct causes until she reaches a set of root causes. The result is the desired "backtrace": a causal chain of events that explains how the observed event came to pass. We refer to this as the *provenance* [1] of the event.

Provenance can be a great tool for debugging complex interactions, but there are cases that it cannot handle. For instance, suppose that the administrator observes that a certain server is *no longer receiving any requests* of a particular type. The key difference to the earlier scenario is that the observed symptom is not a positive event, such as the arrival of a packet, that could serve as a "lead" and point the administrator towards the root cause. Rather, the observed symptom is a negative event: the *absence* of packets of a certain type. Negative events can be difficult to debug: provenance does not help, and even a manual investigation can be difficult if the administrator does not know where the missing packets would normally come from, or how they would be generated.

Nevertheless, it is possible to construct a similar "backtrace" for negative events, using the concept of *negative provenance* [8]. The key insight is to use counterfactual reasoning, i.e., to examine all possible causes that *could have* produced the missing effect. For instance, it might be the

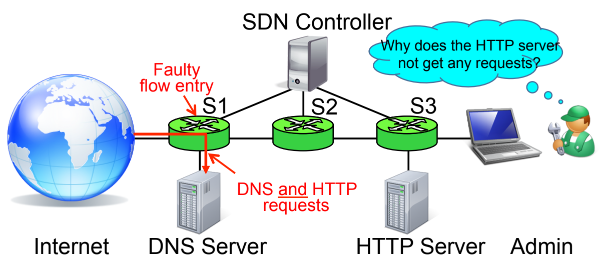


Figure 1: Negative event scenario: Web requests from the Internet are no longer reaching the web server because a faulty program on the controller has installed an overly general rule in the leftmost switch (S1).

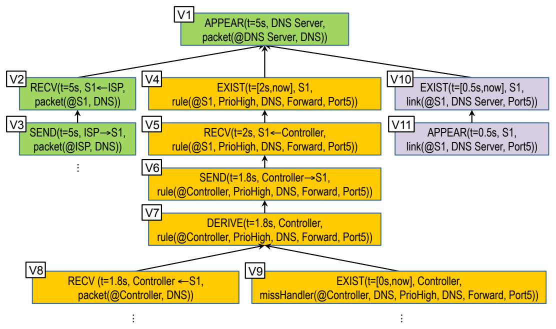

V1 APPEAR(t=5s, DNS Server, packet(@DNS Server, DNS))
V2 RECV(t=5s, S1←ISP, packet(@S1, DNS))
V3 SEND(t=5s, ISP→S1, packet(@ISP, DNS))
V4 EXIST(t=[2s,now], S1, rule(@S1, PrioHigh, DNS, Forward, Port5))
V5 RECV(t=2s, S1←Controller, rule(@S1, PrioHigh, DNS, Forward, Port5))
V6 SEND(t=1.8s, Controller→S1, rule(@Controller, PrioHigh, DNS, Forward, Port5))
V7 DERIVE(t=1.8s, Controller, rule(@Controller, PrioHigh, DNS, Forward, Port5))
V8 RECV (t=1.8s, Controller ←S1, packet(@Controller, DNS))
V9 EXIST(t=[0s,now], Controller, missHandler(@Controller, DNS, PrioHigh, DNS, Forward, Port5))
V10 EXIST(t=[0.5s,now], S1, link(@S1, DNS Server, Port5))
V11 APPEAR(t=0.5s, S1, link(@S1, DNS Server, Port5))

Figure 2: *Positive* provenance example, explaining how a DNS packet made its way to the DNS server.

case that the missing packets could only have reached the server through one of two upstream switches, and that one of them is missing a rule that would match the packets. Based on the controller program, we might then establish that the missing rule could only have been installed if a certain condition had been satisfied, and so on, until we either reach a positive event (such as the installation of a conflicting rule with higher priority) that can be traced with normal provenance, or a negative root cause (such as a missing entry in a configuration file). However, generating negative provenance is substantially more challenging than positive provenance, its dual "twin", e.g., because it involves universal quantifiers (absence of *any* relevant cause) rather than existential quantifiers (presence of a *specific* relevant cause).

Negative provenance could be a useful debugging tool for software-defined networks, but so far it has not been explored very much. A small number of papers from the database community [3, 8, 14] have used negative provenance to explain why a given database query did *not* return a certain tuple, but we are not aware of any previous applications in the networking domain.

This paper is intended as a first step towards a negative provenance system for debugging and forensics in software-defined networks. We define a simple model for negative provenance in this setting, as well as a technique for inferring the provenance of a given negative event, and we describe a number of heuristics that can simplify the resulting "backtrace" to produce more concise explanations. Finally, we present results from a case study to illustrate how negative provenance can help diagnose realistic network problems. We hope that our approach will eventually lead to a provenance-based debugger for SDNs that supports both positive and negative events; we describe some of the remaining challenges in Sections 3.4 and 3.5.

2. OVERVIEW

2.1 Scenario: Network debugging

Figure 1 shows a simple example scenario that illustrates the problem we are focusing on. A network administrator manages a small network that includes a DNS server, a web server, and a connection to the Internet. At some point, the administrator notices that the web server is no longer receiving any requests from the Internet. The administrator strongly suspects that the network is somehow misconfigured, but the only observable symptom is a *negative event* (the absence of web requests at the server), so there is no obvious starting point for an investigation.

Today, the only way to resolve such a situation is to manually inspect the network until some positive symptom (such as requests arriving at the wrong server) is found. In the very simple scenario in Figure 1, this is not too difficult, but in a data center or large corporate network, it can be a considerable challenge. It seems preferable for the administrator to directly ask the network for an explanation of the negative event, similar to a "backtrace" in a conventional debugger. This is the capability we seek to provide.

2.2 Positive provenance

For *positive* events, there is a well-understood way to generate such a "backtrace": whenever an event occurs or some new state is generated, the system keeps track of its causes, and when an explanation is requested, the system recursively explains each event with its direct causes, until it reaches a set of "base events" (such as external inputs) that cannot be explained further. The result can be represented as a DAG, in which each vertex represents an event and each edge a direct causal relationship. In the database literature, this is called the *provenance* [1] of the event; to distinguish it from negative provenance, we will refer to it as *positive provenance*.

Figure 2 shows an example that explains why a DNS request appeared at the DNS server at time $t = 5$ (V1). The DNS server had received the packet from switch S1, which in turn had received it from the Internet (V2–V3); the switch was connected to the DNS server via port #5 (V10–V11) and had a flow entry that directed DNS packets to that port (V4). The flow entry had been installed at $t = 2$ (V5–V7) because the switch had forwarded an earlier DNS packet to the controller (V8), which had generated the rule based on its configuration (V9).

Systems like ExSPAN [21] and SNP [20] can construct this type of positive provenance on demand; it is often a useful debugging tool, just like the "backtraces" that many debuggers offer today.

2.3 Case study: Broken flow entry

We now return to the scenario in Figure 1. One possible reason for this situation is that the administrator has configured the controller to produce a generic, low-priority flow entry for DNS traffic and a specific, high-priority flow entry for HTTP traffic. If both entries are installed, the system works as expected, but if the low-priority entry is installed first, it matches HTTP packets as well; thus, these packets are not forwarded to the controller and cannot trigger the installation of the high-priority entry. This subtle race condition might manifest only at runtime, e.g., when both entries expire simultaneously during an occasional lull in traffic; thus, it could be quite difficult to find.

Provenance would be useful in this situation because it can track causality. A simple "diff" of the network state before and after the time HTTP traffic stopped arriving would return both too much and too little: it would typically show many causally unrelated changes that happened around the same time, but it would probably not show the actual root cause (if the problem resulted from a chain of events that began much earlier). However, *positive* provenance is not helpful here because, as long as requests are still arriving at the HTTP server, their provenance contains only the high-priority entry, and when the requests stop arriving, there is no longer anything to generate the provenance *of*!

2.4 Negative provenance

Although positive provenance cannot explain negative events, there is a way to construct a similar "backtrace" for such events: instead of explaining how an actual event *did* occur, we can simply find all the ways in which a missing event *could have* occurred, and then show, as a "root cause", the reason why each of them did not come to pass.

At first glance, this approach seems infeasible: even in the simple scenario in Figure 1, there are innumerable combinations of flow entries and packet arrivals that could bring requests to the web server. However, *almost all of them will be inconsistent with the SDN controller's program*, which can only produce a few different flow entries, under very specific conditions. This rules out most of the possible explanations.

Moreover, we can use a kind of counterfactual reasoning to recursively generate the explanations, not unlike positive provenance: for a web request to arrive at the web server, a request would have had to appear at the rightmost switch (S3), which did not happen. Such a request could only have come from the switch in the middle (S2), and, eventually, from the switch on the left (S1). But S1 would only have sent the request if there had been 1) an actual request, 2) a matching flow entry with a forward action to S2, and 3) no matching higher-priority flow entry. Conditions 1) and 2) were satisfied, but condition 3) was not (because of the DNS server's flow entry). We can then ask where the higher-priority flow entry came from, which can be answered with positive provenance. We refer to such a counterfactual explanation as *negative provenance*.

2.5 Challenges

The key challenge in working with negative provenance is that its explanations can be far more complex than those from positive provenance. There are two key reasons for this: 1) Unlike positive provenance, which can be conceptually viewed as a subgraph of a *finite* graph that captures the dependencies among all system states and state changes, negative provenance does not have a well-defined model to start with – there is an *infinite* set of negative events! 2) Whenever an effect can occur in more than one possible ways, positive provenance can simply report the *specific* cause, whereas negative provenance must consider *each possible* cause and show for each of them why it did not occur.

In addition, the overwhelming complexity of a naïvely-presented negative provenance might limit its usability. Thus, it is important to simplify the explanations whenever possible, e.g., by hiding branches that represent obvious logical contradictions. In Section 3.4, we describe some simple heuristics that can help to accomplish this.

3. BASIC NEGATIVE PROVENANCE

In this section, we show how to derive a simple provenance graph for NDlog programs that includes both positive and negative events.

3.1 Background: Network Datalog

For ease of exposition, we will assume here that the controller programs are written in *network datalog (NDlog)* [12] because NDlog's declarative syntax makes provenance particularly easy to see. (Our approach does not depend on NDlog, however; we are currently working on a frontend for Frenetic [5].) Below, we briefly review the features of NDlog that are relevant here.

In NDlog, the state of a node (switch, controller, or server) is modeled as a set of *tables*, which each contain a number of *tuples*. For instance, an SDN switch might contain a table called `rule`, and each tuple in this table might represent a flow entry, or a SDN controller might have a table called `packetIn` that contains the packets it has received from the switches. Tuples can be manually inserted, or they can be programmatically derived from other tuples; the former are called *base tuples*, and the latter are called *derived tuples*.

NDlog programs consist of *rules* that describe how tuples should be derived from each other. For instance, the rule `A(@X,P):-B(@X,Q),Q=2*P` says that a tuple `A(@X,P)` should be derived on node X whenever there is also a `B(@X,Q)` tuple on that node, and `Q=2*P`. Here, P and Q are variables that must be instantiated with values when the rule is applied; for instance, a `B(@X,10)` tuple would create an `A(@X,5)` tuple. The `@` operator specifies the node on which the tuple resides. Rules may include tuples from different nodes; for instance, `C(@X,P):- C(@Y,P)` says that tuples in the C-table on node Y should be sent to node X and inserted into the C-table there.

A key advantage of a declarative formulation is that causality is very easy to see: if a tuple `A(@X,5)` was derived using the rule above, then `A(@X,5)` exists simply because `B(@X,10)` exists, and because `10=2*5`.

3.2 Provenance graph

Provenance can be represented as a DAG in which the vertices are events and the edges indicate direct causal relationships. Thanks to NDlog's simplicity, it is possible to define a very simple provenance graph for it, with only seven types of event vertices (based on [20]):

- EXIST$([t_1, t_2], N, \tau)$: Tuple τ existed on node N from time t_1 to t_2;
- INSERT(t, N, τ), DERIVE(t, N, τ): Base (derived) tuple τ was inserted (derived) on node N at time t;
- APPEAR(t, N, τ), DISAPPEAR(t, N, τ): Tuple τ appeared or disappeared on node N at time t; and
- SEND$(t, N_1 \rightarrow N_2, \tau)$, RECEIVE$(t, N_1 \leftarrow N_2, \tau)$: τ was sent (received) by node N_1 to N_2 at time t.

The edges between the vertices correspond to their intuitive causal connections: tuples can appear on a node because they a) were inserted as base tuples, b) were derived from other tuples, or c) were received in a message from another node (for cross-node rules). Messages are received because they were sent, and tuples exist because they appeared. Note that vertices are annotated with the node on which they occur, as well as with the relevant time; the latter will be important for negative provenance because we will often need to reason about past events.

We can extend this model to support negative provenance, by associating each vertex with a negative "twin":

- NEXIST$([t_1, t_2], N, \tau)$: Tuple τ never existed on node N in time interval $[t_1, t_2]$;
- NINSERT$([t_1, t_2], N, \tau)$, NDERIVE$([t_1, t_2], N, \tau)$: τ was never inserted (derived) on N in $[t_1, t_2]$;
- NAPPEAR$([t_1,t_2],N, \tau)$, NDISAPPEAR$([t_1,t_2],N, \tau)$: Tuple τ never (dis)appeared on N in $[t_1, t_2]$; and
- NSEND$([t_1, t_2], N, \tau)$, NRECEIVE$([t_1, t_2], N, \tau)$: τ was never sent (received) by node N in $[t_1, t_2]$.

Again, the causal connections are the intuitive ones: tuples never existed because they never appeared, they never appeared because they were never inserted, derived, or received, etc. However, note that, unlike their positive counterparts, *all* negative vertices are annotated with time intervals: unlike positive provenance, which can refer to specific events at specific times, negative provenance must explain the *absence* of events in certain intervals.

3.3 Graph construction

Given an event vertex, it is possible to mechanically construct the provenance graph that explains it (using a log of the program's execution). We have developed an initial version of such an algorithm, based on an extension of [20], but we cannot present it in detail here due to lack of space. Instead, we briefly describe some of the interesting cases we encountered.

Tuples that never existed: Positive provenance is usually implemented by maintaining "backpointers" that link each tuple to the tuples from which it was derived. This makes it easy to explain the existence or appearance of a tuple. But to explain why a tuple *never* appeared in a certain time interval, we must potentially inspect the log for the entire interval, which can be expensive. However, it should be possible to avoid most of this overhead with a suitable index, which we are currently developing.

Explaining non-derivation: When explaining why a rule with multiple preconditions did not derive a certain tuple, we must consider a potentially complex parameter space. For instance, if `A(@X,p):-B(@X,p,q,r),C(@X,p,q)` did not derive `A(@X,10)`, we can explain this with the absence of `B(@X,10,q,r)`, `C(@X,10,q)`, or a combination of both – e.g., by dividing the possible q values between the two preconditions. Different choices can result in explanations of dramatically different sizes once the preconditions themselves have been explained; hence, we would like to choose a partition of the parameter space (here, $Q \times R$) that results in the smallest possible explanation. In general, finding such a partition is at least as hard as the SetCover problem, which is NP-hard; however, we have found that simple "greedy" heuristics seem to work quite well in practice. We are currently refining these heuristics further.

Missing messages: A similar challenge exists when explaining why a certain message m was not received. Theoretically, *any* node in the system could have sent m, so it may seem as if we had to give a specific reason for each of them. But in practice, most protocols restrict how communication can occur; for instance, in routing protocols, messages can typically come only from direct neighbors. Moreover, it is often the case that a given message can only come from a *specific set* of neighbors: for instance, a BGP message whose AS path starts with 55 should come from a router in AS 55. However, to avoid giving misleading answers, we must only rule out a possible sender if the program clearly prohibits it; we are currently investigating the use of static analysis to infer such restrictions.

3.4 Pruning unhelpful branches

"Raw" negative provenance graphs constructed according to Section 3.3 are correct and complete, but they are sometimes cluttered with explanations that are technically correct but not very helpful. We have identified two techniques that can safely remove most of the clutter.

Early termination: Some explanations contain logical inconsistencies: for instance, if the absence of a tuple τ_1 with parameter space S_1 might be explained by the absence of a tuple τ_2 with parameter space $S_2 \subseteq S_1$, or if a derivation is only possible when a variable is out of range. If we can recognize such inconsistencies early, there is no need to

```
s0: pktTemp(@S,H,Pr) :- packet(@S,H),highestPriority(@S,Pr).
s1: drop(@S,H) :- pktTemp(@S,H,Pr),rule(@S,Pr,M,A,Pt),H==M,A=='Drop'.
s2: packet(@D,H) :- pktTemp(@S,H,Pr),rule(@S,Pr,M,A,Pt),H==M,A=='Fwd',link(@S,D,Pt).
s3: ruleMatch(@S,H,Pr,a_Count<*>) :- pktTemp(@S,H,Pr),rule(@S,Pr,M,A,Pt),H==M.
s4: pktTemp(@S,H,Pr') :- ruleMatch(@S,H,Pr,C),C==0,Pr':=Pr-1,Pr>=0.
s5: rule(@S,Pr,M,A,Pt) :- pktTemp(@S,H,Pr),rule(@S,Pr,M,A,Pt),H==M.
s6: packetIn(@C,H,S) :- pktTemp(@S,H,Pr),rule(@S,Pr,M,A,Pt),H==M,Pr==0,controller(@S,C).
s7: drop(@S,H) :- pktTemp(@S,H,Pr),Pr==-1.
c0: rule(@S,Pr,M,A,Pt) :- packetIn(@C,H,S),missHandler(@C,H,Pr,M,A,Pt),ofswitch(@C,S).
```

Figure 3: NDlog model of an SDN: When a packet arrives, it is matched against flow entries, starting with the highest-priority entry (s0). If an entry matches, its action is applied (s1+s2), and the entry is refreshed (s5), otherwise lower-priority entries are tried (s3+s4); the lowest-priority entry sends packets to the controller (s6). If no entry matches, the packet is dropped (s7). When the controller receives a packet, it computes an entry and sends it to the switch (c0).

continue generating the provenance until a set of base tuples is reached, since the precondition is clearly unsatisfiable. Thus, we can safely truncate the corresponding branch of the provenance tree.

Avoiding redundancy: Sometimes the same fact is needed at multiple points in the graph. To avoid redundancy, we include the explanations of such facts only once.

Application-specific invariants: Some explanations may be irrelevant for the particular SDN that is being debugged. For instance, certain data – such as constants, topology information, or state from a configuration file – changes rarely or never, so the absence of changes does not usually need to be explained. If necessary, the programmer can briefly annotate such state, so the system can omit further explanations when this state has not changed.

On-demand exploration: If the graph is large, the human investigator can initially be shown only part of it, perhaps up to a certain "depth". The investigator can then expand additional vertices interactively, as needed.

3.5 Improving readability

Another way to make negative provenance graphs more useful for the human investigator is to display the provenance at different levels of detail. For instance, if a message fails to appear at node N_1 but could only have originated at node N_2 several hops away, the basic provenance tree would show, for each node on the path from N_1 to N_2, that the message was not sent from there, because it failed to appear there, because it was not received from the next-hop node, etc. We can improve readability by summarizing these (thematically related) vertexes into a single *super-vertex*. If the human investigator requests additional details, the fine-grained vertexes can still be shown.

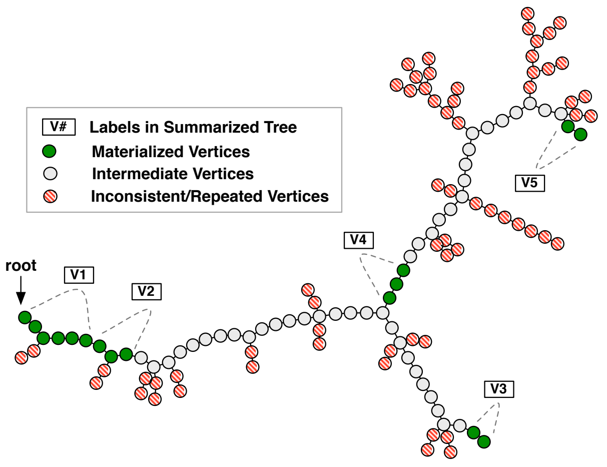
So far, we have identified three situations where this summarization can be applied. The first is a chain of transient events that originates at one node and terminates at another, as in the above example; we replace such chains by a single super-vertex. The second is the (common) sequence NEXIST$([t_1,t_2],N,\tau)$ $\leftarrow$ NAPPEAR$([t_1,t_2],N,\tau)$ $\leftarrow$ NDERIVE$([t_1,t_2],N,\tau)$, which says that a tuple was never derived; we replace this with an ABSENCE$([t_1,t_2],N,\tau)$ super-vertex; its positive counterpart EXISTENCE$([t_1,t_2],N,\tau)$ is used to replace a positive sequence. The third situation is a derivation that depends on a small set of triggers – e.g., flow entries can only be generated when a packet p is forwarded to the controller C. In this case, the basic provenance will contain a long series of NAPPEAR$([t_i,t_{i+1}],C,p)$ vertices that explain the *common* case where the trigger packet p *does not* exist; we replace these with a single super-vertex ONLY-EXIST$(\{t_1,t_2,\ldots\}\in[t_{\text{start}},t_{\text{end}}],C,p)$ that initially focuses attention on the *rare* cases where the trigger *does* exist. Again, the full explanation can still be shown on demand.

4. CASE STUDY

We have implemented an early prototype of a negative provenance engine for SDNs, based on the algorithm and the heuristics in Section 3. To illustrate the provenance results our engine produces, we have set up the scenario from Figure 1 and Section 2.3 (our running example) in a simulator, using an NDlog model of an OpenFlow switch, which is shown in Figure 3 and explained briefly in the caption of that figure. (Recall that we use NDlog only for ease of explanation; the switch does not actually have to run NDlog.) The query we have motivated earlier (why HTTP requests are no longer appearing at the web server) corresponds to a NAPPEAR$([t_1,t_2],$ $D,$ packet$(@D,$ HTTP$))$ in this model, where $[t_1,t_2]$ is an interval during which the administrator has not observed any web requests.

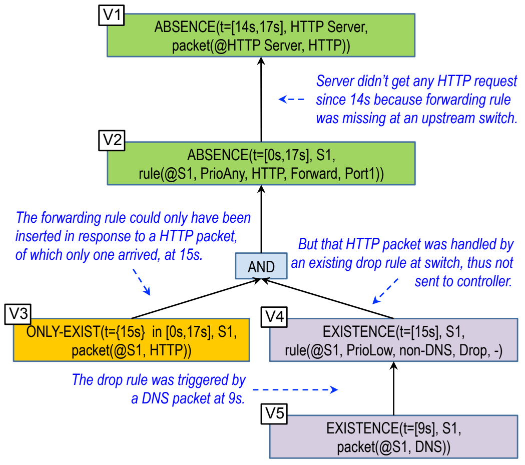
Figure 4 shows the "raw" negative provenance, without the heuristics from Sections 3.4 and 3.5. The details are not shown, but the tree is clearly too large to be used directly by a human investigator. However, many of the smaller branches (shown in red) represent logical inconsistencies and can be pruned, while many of the vertices in the main branch (shown in gray) represent technicalities, such as event chains across multiple nodes, which can be summarized.

Figure 5 shows the final negative provenance after applying our heuristics. This explanation is much more readable: HTTP requests did not arrive at the HTTP server (V1) because there was no suitable flow entry at the switch (V2). Such an entry could only have been installed if a HTTP packet had arrived (V3) and caused a table miss, but the latter did not happen because there already was an entry (the low-priority DNS entry, V4) that matched HTTP packets,



V# Labels in Summarized Tree
● Materialized Vertices
○ Intermediate Vertices
▨ Inconsistent/Repeated Vertices
root
V1
V2
V3
V4
V5

Figure 4: *Raw* negative provenance tree for our example query from Figure 1 (absence of HTTP requests).



V1 ABSENCE(t=[14s,17s], HTTP Server, packet(@HTTP Server, HTTP))
Server didn't get any HTTP request since 14s because forwarding rule was missing at an upstream switch.
V2 ABSENCE(t=[0s,17s], S1, rule(@S1, PrioAny, HTTP, Forward, Port1))
The forwarding rule could only have been inserted in response to a HTTP packet, of which only one arrived, at 15s.
AND
But that HTTP packet was handled by an existing drop rule at switch, thus not sent to controller.
V3 ONLY-EXIST(t={15s} in [0s,17s], S1, packet(@S1, HTTP))
V4 EXISTENCE(t=[15s], S1, rule(@S1, PrioLow, non-DNS, Drop, -)
The drop rule was triggered by a DNS packet at 9s.
V5 EXISTENCE(t=[9s], S1, packet(@S1, DNS))

Figure 5: Final provenance tree for our example query, after the techniques from Section 3.4 and 3.5 have been applied.

and that entry had been installed in response to an earlier DNS packet (V5). We believe that "backtraces" of this kind would be useful in debugging complex problems in SDNs.

Note that our heuristics are not specific to this scenario, so we can hope to achieve a comparable reduction in complexity for other queries. We have confirmed this for several other scenarios, but we omit the results for lack of space.

5. RELATED WORK

Network debugging: Many tools and techniques for network debugging and root cause analysis have been proposed, e.g., [4, 7, 13], but most focus on explaining positive events. Hubble [10] uses probing to find AS-level reachability problems but is protocol-specific; header space analysis [11] provides finer-grain results but relies on static analysis and thus cannot explain complex, dynamic interactions like the ones we consider here. ATPG [19] tests for liveness, reachability, and performance, but cannot handle dynamic nodes like the SDN controller. NICE [2] uses model checking to test whether a given SDN program has specific correctness properties; this approach is complementary to ours, which focuses on investigating the cause of unforeseen problems at runtime. We are not aware of any protocol-independent systems that can explain negative events in a dynamic distributed system.

Negative provenance: There is a substantial literature on tracking provenance in databases [1, 6, 9, 15, 18] and in networks [20, 21], but only a few papers consider negative provenance. Huang et al. [8] and Meliou et al. [14] focus on instance-based explanations for missing answers, that is, how to obtain the missing answers by making modifications to the value of base instances (tuples). Why-Not [3] and ConQueR [16] provides query-based explanations for SQL queries, which reveal over-constrained conditions in the queries and suggest modifications to them. None of these papers consider distributed environments and networks, as we do here.

6. SUMMARY AND CURRENT STATUS

In this paper, we have argued that negative provenance is a promising approach to a particularly difficult class of diagnostic problems: the cases where an expected event *fails to occur* and the human administrator is left with no starting point for her investigation. Negative provenance enables the administrator to ask "Why not?" queries and to obtain the equivalent of a backtrace in response, much like those in a traditional debugger.

At first glance, it may seem that negative provenance cannot work: there seem just too many ways in which a missing event *could* have occurred. However, we have provided initial evidence that, with careful program analysis and with appropriate summarization and pruning, negative provenance can produce surprisingly simple, high-quality explanations. Thus, further work on a more complete system for negative provenance seems warranted.

However, much is left to be done before we can reach this goal. For instance, negative provenance requires a runtime system for storing execution traces, similar to ExSPAN [21] or SNP [20]; front-ends for popular SDN programming languages like Frenetic [5] should be added; suitable index structures for fast query processing need to be developed; and the summarization and pruning heuristics need to be further refined. We are pursuing these goals in our ongoing work.

Acknowledgments

We thank the anonymous reviewers for their helpful comments and suggestions. We also thank Alexander J. T. Gurney for helpful comments on an earlier draft of this paper. This work was supported by NSF grants CNS-0845552, CNS-1040672, CNS-1054229 and CNS-1065130. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the funding agencies.

7. REFERENCES

[1] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *Proceedings of the 8th International Conference on Database Theory (ICDT)*, Jan. 2001.

[2] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE way to test OpenFlow applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Apr. 2012.

[3] A. Chapman and H. V. Jagadish. Why not? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2009.

[4] A. Feldmann, O. Maennel, Z. M. Mao, A. Berger, and B. Maggs. Locating Internet routing instabilities. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2004.

[5] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming (ICFP)*, Sept. 2011.

[6] T. J. Green, G. Karvounarakis, N. E. Taylor, O. Biton, Z. G. Ives, and V. Tannen. ORCHESTRA: Facilitating collaborative data sharing. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2007.

[7] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. Where is the debugger for my software-defined network? In *Proceedings of the 1st Workshop on Hot Topics in Software Defined Networking (HotSDN)*, Aug. 2012.

[8] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *Proceedings of the VLDB Endowment*, 1(1):736–747, Aug. 2008.

[9] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2011.

[10] E. Katz-Bassett, H. V. Madhyastha, J. P. John, A. Krishnamurthy, D. Wetherall, and T. Anderson. Studying black holes in the Internet with Hubble. In *Proceedings of the 5th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Apr. 2008.

[11] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Apr. 2012.

[12] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Communications of the ACM*, 52(11):87–95, Nov. 2009.

[13] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2011.

[14] A. Meliou and D. Suciu. Tiresias: The database oracle for how-to queries. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, May 2012.

[15] C. Ré, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*, Apr. 2007.

[16] Q. T. Tran and C.-Y. Chan. How to ConQueR why-not questions. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2010.

[17] M. Welsh. What I wish systems researchers would work on. Blog post, `http://matt-welsh.blogspot.com/2013/05/what-i-wish-systems-researchers-would.html`, May 2013.

[18] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2005.

[19] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, Dec. 2012.

[20] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[21] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at Internet-scale. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2010.