Causal Network Telemetry

Yunhe Liu
yunheliu@cs.cornell.edu
Cornell University
Ithaca, NY, USA

Nate Foster
jnfoster@cs.cornell.edu
Cornell University
Ithaca, NY, USA

Fred B. Schneider
fbs@cs.cornell.edu
Cornell University
Ithaca, NY, USA

ABSTRACT

Current approaches to network observability rely on techniques like active probing, packet sampling, and path-level telemetry, which only provide a partial view. This paper presents causal telemetry, a new model that adapts ideas from distributed systems to the network setting. Causal telemetry captures causal relationships between events, including those that take place on physically separated devices. We motivate causal telemetry through examples, we show how it can be used to diagnose anomalies and faults, and we present algorithms for constructing the needed causal graphs from network executions. We develop a P4-based prototype implementation, CoCaTel, and discuss a case study that uses causal telemetry to detect Priority-Based Flow Control (PFC) deadlocks.

CCS CONCEPTS

• **Networks** → **Network monitoring**; **Network management**.

KEYWORDS

Causal analysis; network debugging; P4; programmable networks.

ACM Reference Format:
Yunhe Liu, Nate Foster, and Fred B. Schneider. 2022. Causal Network Telemetry. In *P4 Workshop in Europe (EuroP4 '22), December 9, 2022, Roma, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3565475.3569084

1 INTRODUCTION

Networks have become complex distributed systems [12]. The processors in these distributed systems are switches that use sophisticated algorithms (e.g., [1, 19]) to optimize the paths that packets take as they traverse the network. But complex distributed systems are notoriously difficult to manage, and networks are no exception.

Management of a complex system invariably requires visibility into the states of the system's components. With *in-band network telemetry* (INT) [14, 17], that visibility is provided by augmenting each packet with a log that records how that packet is processed at each hop. These logs are then aggregated at a central collector and the operator gets visibility into the network's operation by querying the collector. For example, a query might retrieve a packet's end-to-end forwarding path, the specific forwarding entries used on individual devices, or the amount of time spent queueing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroP4 '22, December 9, 2022, Roma, Italy
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9935-7/22/12…$15.00
https://doi.org/10.1145/3565475.3569084

The overhead associated with implementing INT is non-trivial. Even so, it has not dissuaded the research community from investigating approaches based on telemetry, doubtless because the visibility provided into a network's operation is so valuable to operators. Visibility, however, depends on what information is collected and what queries are supported.

For the algorithms that network switches run today, queries involving causal connections between events at switches would be invaluable. For example: was a recent change to a forwarding rule at one switch responsible for the delays at another switch, or was it instead the delays themselves that prompted the change? Prior work on supporting INT has largely ignored causality. Reading real-time clocks at switches sometimes works as a proxy, but not always. Moreover, even were clocks kept synchronized, there would still be unnecessary complexities in formulating queries, since packets usually visit switches at different times, not contemporaneously.

This paper introduces a new basis for INT, *causal telemetry*, as well as a prototype, CoCaTel, that establishes the practicality of the approach. CoCaTel supports queries about causality between network events; it also handles all of the queries that other INT systems support. The overhead associated with running CoCaTel is comparable to the overhead of the prior INT systems. Engineering innovations designed to improve the performance of INT implementations should apply to CoCaTel, too.

A proposal to use causality as the foundation for understanding network behavior should not be surprising. Over four decades ago, Lamport [5, 25] showed the benefits of viewing executions of a distributed system in terms of causally-related events, with computations depicted as *space-time diagrams* and with *cuts* used to describe states that might be observed. Causal telemetry uses space-time diagrams as its data model, and CoCaTel incorporates algorithms for constructing and querying this data model.

Existing network telemetry does not track causality.[1] We conjecture two reasons for this gap:

- A misplaced fear that the additional expressiveness causality provides is useless. If the data plane implements mostly-stateless forwarding, why bother tracking inter-packet relationships? The examples in §2 and §6 show that causality is useful for visibility into the operation of algorithms found in today's networks.
- A misplaced fear that tracking causality would be too costly. We find that CoCaTel imposes only modest overheads above INT, which is widely deployed.

In sum, we propose a new foundation for understanding network behavior—causality, rather than real-time clocks. We argue that causality is necessary to answer queries of practical interest in modern networks, including functional and performance queries. We relate causal telemetry to prior work on INT and to systems like

[1] Cuts, however, have been used—see work by Yaseen et al. [37]

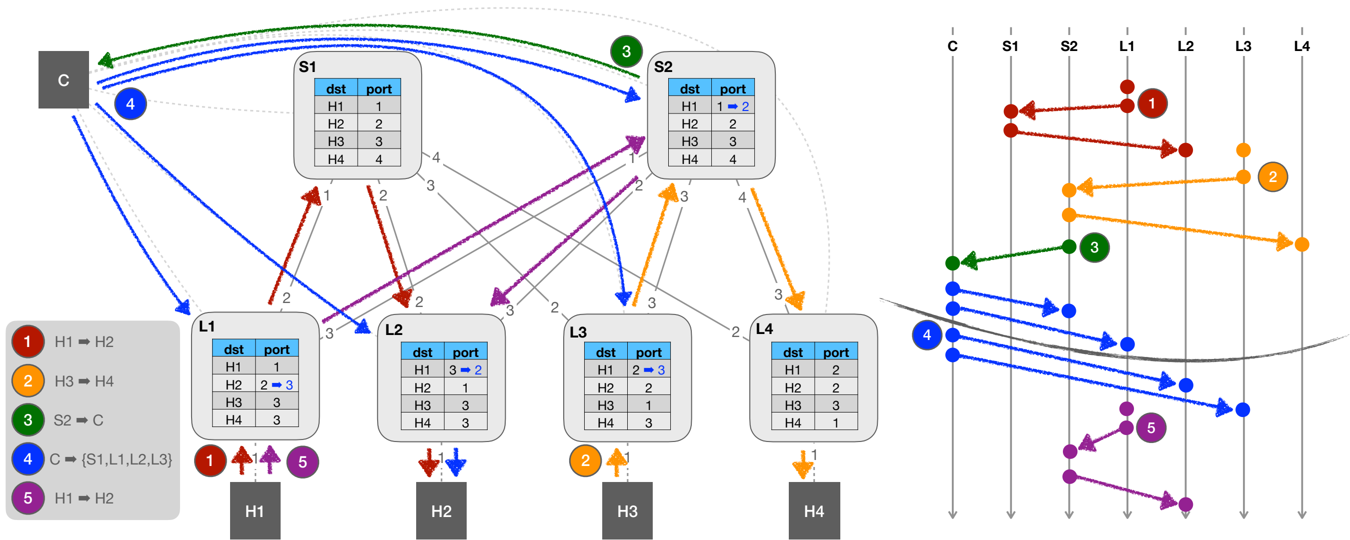

Figure 1: Left: example topology with high latency and transient loop; Right: corresponding space-time diagram.

synchronized snapshots, where a request can be issued to collect current state information. Our CoCaTel prototype demonstrates that causal telemetry can be implemented on P4-based switches. We also discuss opportunities for optimization that can reduce overhead. Finally, we discuss our vision for future work, including support for multi-tenancy and a security model.

2 OVERVIEW

We start by reviewing techniques for understanding network behavior and show how causality can improve on these approaches.

Command-Line Utilities and Packet Sampling. Many operators rely on command-line tools (e.g., `ping` and `traceroute`) to understand network behavior. These tools use probe packets to check connectivity and to determine forwarding paths. "Bump in the wire" monitoring middleboxes (e.g., sFlow, etc.) produce aggregate statistics by sampling packets flowing across links. These approaches are useful, but provide only limited visibility.

Global States. In a software-defined network (SDN), the controller maintains a global view of the network. This view can be analyzed to detect and diagnose faults. For example, data-plane verification tools like Header Space Analysis (HSA) [20] and Veriflow [21], check key invariants (e.g., reachability, loop freedom, etc.) every time the controller changes the network configuration. Recent work by Yaseen et al. [37] uses Chandy-Lamport snapshots [5] of the data plane state. However, while it's attractive to think in terms of global states, this approach also has limitations: a packet traversing the network can be part of multiple global states—e.g., during reconfiguration. Hence, conclusions obtained by analyzing global states are not always valid for states "seen" by individual packets—analyses that examine multiple states are required.

In-Band Telemetry. With INT [14, 17], each packet logs the states it encounters as it traverses the network. For example, the log might record the end-to-end path, the forwarding rules used on each switch, or the total time spent queueing. At the network edge, the log is collected at a server where it can be analyzed. INT is popular, and for good reason—observing how each packet is processed is a powerful capability. But is INT the ultimate network observability tool? Being based on logs for individual packets, it cannot provide insights about scenarios involving multiple packets or the controller. Again, more complex analyses are required.

Our Approach: Causal Telemetry. This paper presents a new approach for understanding network behavior. We call it *causal telemetry*. As the name suggests, causal telemetry is based on INT extended with information to track causality. That is, packet logs aggregate state along the end-to-end forwarding path and they also keep track of causal relationships among packets, control messages, etc. Causal telemetry thus captures aspects of network behavior that are not adequately modeled by existing frameworks.

Example. To illustrate causal telemetry's benefits, consider the execution shown in Figure 1. The network has a leaf-spine topology with two spines (**S1** and **S2**), four leaves (**L1** through **L4**), and four hosts (**H1** through **H4**). The controller (**C**) manages the configuration of the switches, which initially forward packets along these paths (among others):

- **H1** to **H2** via [**L1**,**S1**,**L2**]
- **H2** to **H1** via [**L2**,**S2**,**L1**]
- **H3** to **H1** via [**L3**,**S1**,**L1**]
- **H3** to **H4** via [**L3**,**S2**,**L4**]

Suppose the following sequence of events occurs, as depicted in Figure 1. First, **H1** sends a packet to **H2** (①). Second, **H3** sends a packet to **H4** (②). Third, after receiving the packet from **H3**, switch **S2** signals the controller (③), which reconfigures the switches to forward packets along the following paths, among others (④):

- **H1** to **H2** via [**L1**, **S2**, **L2**]
- **H2** to **H1** via [**L2**, **S1**, **L1**]
- **H3** to **H1** via [**L3**, **S2**, **L2**, **S1**, **L1**]

Finally, **H1** sends another packet to **H2**, which now follows a different path (⑤). The reasons for signaling the controller are unimportant—e.g., perhaps **S2**'s was experiencing congestion or its existing forwarding rules timed out. What matters is that the network was reconfigured.

Now suppose the final packet (⑤) experienced high latency at **S2**. Using INT, we could easily detect this latency increase by recording the amount of time the packet spends in each queue [14]. However, it's surprisingly hard with INT to get a deeper explanation for why the performance of the **H1** to **H2** flow was degraded. If the logs have detailed timestamps, we can determine what other packets were in the queue on **S2** when the packet experienced high latency. But we cannot easily explain why **H1** to **H2** traffic was being forwarded via **S2** in the first place. For reaching that conclusion, we would need to relate the packet coming from **H1** to the packet coming from **H3** that caused **S2** to signal the controller! Note that because the packets followed disjoint paths, we cannot reliably determine which came first, unless the switches have synchronized clocks.

Here, what we need is not only to keep track of the states observed by individual packets, but also to record causal relationships between them. Figure 1 shows a space-time diagram that gives those causal relationships for this example. The full interpretation of this diagram is explained in the next section. For now, notice that it records the trajectories followed by each individual packet, as well as relationships among packets. The re-routing of the **H1** to **H2** flow after the earlier packet from **H3** to **H4** can be easily seen. Hence, the **H3** packet is a potential cause of that routing change.

There is a second issue arising in this example that can also be understood in terms of the space-time diagrams. To reconfigure the switches, the controller sent messages to **S1** and **L1** through **L3** concurrently. However, if **S2** is reconfigured before **L2**, then there will be a transient loop—**L2** will use the original configuration to send traffic destined for **H1** to **S2**, while **S2** will use the new configuration to send traffic destined for **H1** back to **L2**. This loop disappears as soon as the configuration on **L2** is updated. But this behavior is likely a bug and could indicate a deeper issue with the controller. With INT, we would not detect this bug, since none of the packets in our scenario followed a loopy path. But using the space-time diagram, we immediately see the bug is possible—e.g., if an **H2** to **H1** packet had been sent at the "cut" indicated by the black line, the packet would have been forwarded in a loop. So we see how a space-time diagram can be used for hypothetical reasoning and post-hoc analysis too.

3 BUILDING SPACE-TIME DIAGRAMS

Causal telemetry supports three kinds of network events: sends, receives, and internal events. Send and receive events result from communication between different switches, or between switches and the controller; internal events result from packet-processing on individual devices.

Like INT [14], causal telemetry generates a log for each network event—e.g., receipt of a packet at a switch—and collects these logs at a server, where they can be analyzed. Existing systems [17] have already shown the feasibility of generating such logs, typically by using a custom header to aggregate information along the end-to-end forwarding path.

However, to capture causality between events—especially those occurring on physically-separate nodes—we need to add additional information to the log. We introduce an event identifier (*event_id*), which is a pair comprising a node identifier (*node_id*) and a monotonically increasing local timestamp (*node_ts*). For internal events, we simply store the event identifier in the log. For send events, we store the event identifier in the log and also embed it in the outgoing message. For receive events, we extract the identifier of the sender and record the sender and receiver in the log.

This simple scheme suffices for building a space-time diagram—i.e., a directed graph in which vertices encode events and edges encode causality between events. When we draw a space-time diagram, we group the vertices representing events on the same device (e.g. switch or controller) together.[2]

It is easy to order events on the same device, as event identifiers are unique and have monotonically increasing timestamps. Hence, timestamps induce a total order on the events on a given device. However, thus far, event identifiers do not impose an order on events occurring on different devices. It turns out that not every pair of events on different devices can be ordered. The best one can hope to do, as observed by Lamport [25], is to relate send and receive events—i.e., the send event causally precedes the receive event. This is precisely the information stored in the logs of the receive events. By reading the log of each receive event, we are able to add the causal edges between every pair of send event and receive event to the space-time diagram.

Readers familiar with the distributed systems literature may observe that this scheme is not a direct implementation of the Lamport clocks. Indeed, because our scheme explicitly carries event identifiers with packets, we are able to encode the edges in the causal diagram directly, rather than having to reconstruct them from an ordering on logical clock timestamps.

4 P4-BASED IMPLEMENTATION

Figure 2 depicts a P4 switch that generates the logs needed to implement causal telemetry, as just described.

Generating Logs. Log data is sourced from ordinary packet headers; metadata (managed by the user program); and by a special causal telemetry header, which keeps track of the (*event_id*). Log packets are generated using the E2E clone primitive on P4 switches. After the log packet is generated, it traverses the egress pipeline, where it is truncated to remove extraneous information—e.g., the payload. The log packet is then forwarded to the collector.

To reduce the number of logs that must be transmitted, causal telemetry optionally supports batching: P4 registers are used to store the information from several logs, and the combined log is shipped periodically to the collector (e.g. when the size of the combined log approaches the MTU). Note that this approach is slightly different than the one used in INT systems, where logs are aggregated on packets and only sent to the collector at the last hop. Batching event logs on switches rather than in packets provides an easy mechanism for enabling or disabling telemetry—an operator can use an active packet to turn log collection "on" or "off". Batch

[2]Devices can be defined at different granularities as long as the events on the same device share the same local clock. For example, we might choose to model a device as a single pipeline on a switch, or as the entire switch.

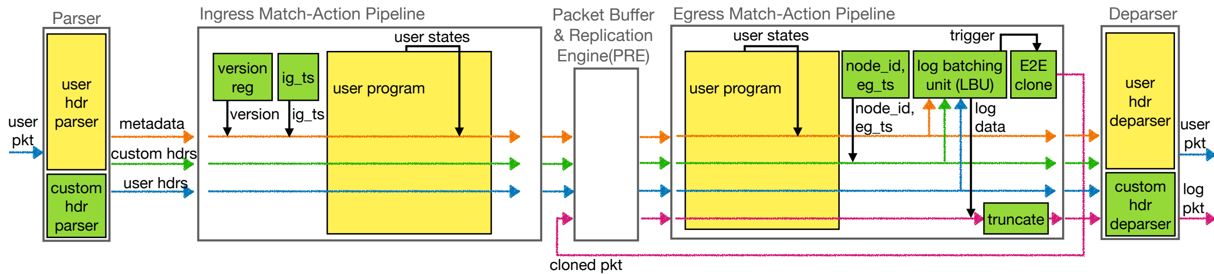

Figure 2: A packet processed by a BMv2 switch implementing causal telemetry.

logs on switches also lays the groundwork for on-switch processing of logs (e.g., filtering, aggregation, etc.), which could further reduce storage and computation costs at the collector.

Device Granularity. Existing INT systems often abstract each switch as a single processor. Real-world switches have multiple independent processing pipelines that operate in parallel. To accurately model the concurrency that may occur within a switch, our prototype implementation models switches at pipeline granularity. When a user packet is processed at a BMv2 switch, we add separate send and receive events to the space-time diagram for the ingress and egress pipelines. Modeling causality at pipeline granularity gives additional visibility—e.g., it allows operators to observe packet reordering that can occur due to queueing.

Control Plane Events. To model causality induced by the control-plane, we add events for messages that travel between controllers and switches. The controller is responsible for generating logs for control messages and delivering the logs to the collector. However, we also need to determine the relative ordering of control events and packet events on switches. To achieve this, we introduce an additional register in the P4 program that records a monotonically increasing version number on receipt of each message from the controller. We store this version number in the log for packet events and extend the ordering on events so that a controller message whose version is less than or equal to V causally precedes all data plane events whose log contains version V. This scheme neatly solves the problem of ordering control and packet events, but it does require atomic switch reconfiguration—i.e., updating the version number and a forwarding rule. In settings where the control API does not support atomic updates, their effect can be simulated using consistent updates [34].

5 COCATEL PROTOTYPE

We have built a P4-based prototype of causal telemetry, called CoCaTel, that runs on the Behavioral Model Version 2 (Bmv2) [9] software switch. CoCaTel consists of ~1100 lines of P4 code to generate and collect event logs in the programmable data plane and ~700 lines of Python code to build the space-time diagram at the centralized server. In addition, we have implemented 6 applications for CoCaTel, including an application that implements all of the query operators in Marple [32], thereby demonstrating that CoCaTel is more expressive.

6 CASE STUDY: PFC DEADLOCK

As a case study to illustrate the power of causal telemetry, we have used CoCaTel to provide visibility into Priority-Based Flow Control (PFC) [13], a mechanism used in the implementation of lossless fabrics. When the queue depth on a switch S exceeds a pre-set threshold, S sends a *PFC_STOP* message to its upstream neighbor, requesting that it stop sending packets to S. Later, if the queue depth at S goes below a pre-set threshold, S sends a *PFC_RESUME* message, informing the upstream switch it may resume.

PFC can create cyclic buffer dependencies between switches, leading to a deadlock [36]. While prior work has developed solutions for deadlock detection in networks [28, 35, 36], these approaches are task specific and cannot be used to monitor other network behaviors at the same time. Existing INT systems [14, 17] have not been used to detect deadlocks, but even in situations where they could help, they would lack the capability to help operators to ascertain the causal sequence of events that led to deadlock in the first place.

In contrast, we can easily detect PFC deadlocks in CoCaTel simply by finding a consistent cut in which there is a cycle of switches, all waiting for their downstream neighbor. CoCaTel's space-time diagram captures the causal ordering of the events that led to the deadlock, which gives operators a basis for identifying good strategies to avoid future occurrences.

Figure 3 shows a scenario where a PFC deadlock exists. There are three concurrent flows:

- **F1** flows from **H1** to **H2** via [**S1**, **S2**]
- **F2** flows from **H2** to **H3** via [**S2**, **S3**]
- **F3** flows from **H3** to **H1** via [**S3**, **S1**]

The relevant switch state, as shown in Figure 3 are (i) the forwarding table; (ii) a register that records whether a *PFC_STOP* has been received from upstream switches, and (iii) the packet queues.

Suppose a packet goes from **S1** to **S2**. Upon receiving this packet, the queue on **S2** exceeds the pre-set threshold and sends a *PFC_STOP* message to **S1**. **S1** then adds **S2** to its PFC register and stops sending packets toward **S2**. The next packet to be dequeued at **S1** is a packet in flow **F1** which, according to **S1**'s forwarding table, should be forwarded to **S2**. But **S1** cannot transmit the packet until it receives a *PFC_RESUME* message from **S2**. So **S1** is "waiting for" **S2** to make progress. In addition, suppose the same interaction subsequently happened between **S2** and **S3**, and between **S3** and **S1**, causing **S2** to wait for **S3** and **S3** to wait for **S1**. It should be clear at this point there is a cyclic "waits-for" dependency among **S1**, **S2** and **S3**, so the network is deadlocked and no switch can make progress.

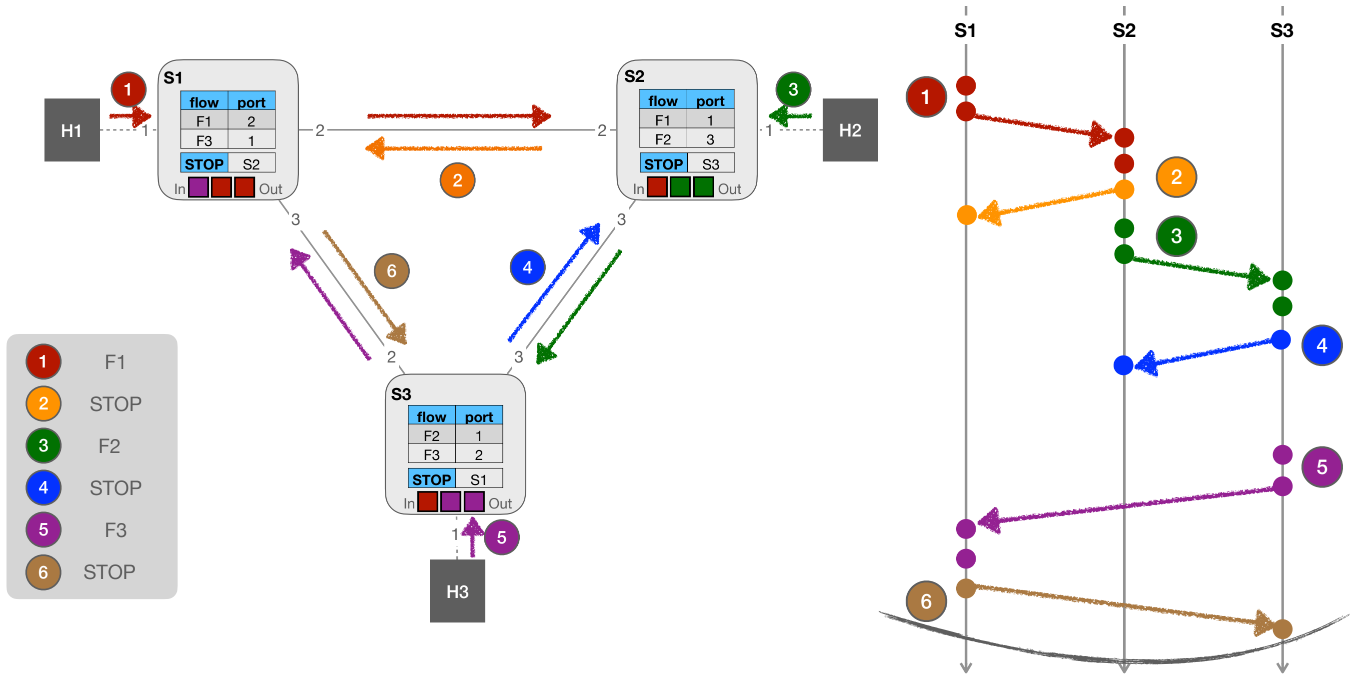

Figure 3: Case study: PFC deadlock detection.

Detecting deadlock is a classic use for consistent cuts on space-time diagrams. The right side of Figure 3 shows a cut, indicated by the black line, in which an operator can easily spot the cyclic wait-for relationships among **S1**, **S2**, and **S3**. Moreover, the space-time diagram clearly shows the order of switches "locked" by the *PFC_STOP* messages is: **S1**, **S2**, and then **S3**. Using this information, an operator can easily locate where the PFC deadlock started in the network and design the resolution accordingly.

Note that PFC deadlock is an example of a purely data-plane algorithm where causality is useful for understanding network behavior. As more and more sophisticated data plane algorithms are implemented in P4, we expect that similar examples will follow.

7 DISCUSSION AND FUTURE WORK

Expressiveness. Causal telemetry is more expressive than existing systems, as can be seen from the fact that the their data models can be obtained as projections of causal telemetry's space-time diagrams. For example, by retaining only selected events of interest, we get the data model used by command-line tools (e.g., `ping` and `traceroute`) and systems based on sampling (e.g., sFlow). By retaining only the events for a single switch, ordered by local timestamp, we get the data model used by Sonata [15]. By grouping events related to each individual flow (e.g., identified by a 5-tuple), ordering them according to causality, and applying various aggregation functions, we get the data model used by sketch-based telemetry systems [7, 11, 26, 27, 31, 33, 38]. Finally, by enumerating consistent cuts we get consistent global states, which is the data model used by Speedlight [37]. Based on these examples and others, we believe causal telemetry is both expressive and the right foundational model for network observability.

Efficiency. Causal telemetry adds only modest overheads to existing INT systems [14, 17]. Specifically, overheads are added at two places. First, *event_id* and control plane version number (described in section 4) are added to each log packet. The added fields do not increase the number of log packets and only constitute a small fraction of each log packet. Causal telemetry also supports log batching to further reduce the overhead of shipping logs to the collector. Since we batch logs on each switch, in the future, we expect that users will be able to make trade-offs between on-switch and at-collector log processing flexibly. Second, we generate logs for messages that travel between controllers and switches whereas existing INT systems do not keep track of control messages at all. The logs for control messages add only modest overheads because the number of control messages is very small compared to the number of data plane packets and these logs can be batched at the controller to further reduce overhead. In the future, we plan to deploy CoCaTel to a hardware testbed and evaluate it on real traffic to conduct an empirical analysis of overheads.

Dynamic Customization. A practical implementation of causal telemetry must navigate trade-offs between expressiveness and overhead. In principle, all of the information related to an event could be written to the log, but in practice, we often need to bound the size of the log to reduce storage and communication overheads. We believe operators should have the power to define what information goes into the log. We imagine a future implementation in which new logging schemes can be defined dynamically, where multiple schemes can be supported concurrently, and where every node in the network—e.g. switches, controllers, ends hosts, etc.—can generate event logs, giving operators visibility into behaviors at different layers.

Failures. Failures can undermine the visibility offered by causal telemetry. For instance, dropped packets may not generate events, and logs may be lost on their way to the collector. These issues arise in systems based on INT as well. To handle dropped packets, one can augment our model with new events. We plan to explore using in-network failure detection and mirror-on-drop to support these extensions. In addition, to handle lost logs, one can create a reliable network fabric for communication with the controller, possibly by assigning higher priority to log packets.

Security. The ability to make detailed observations of network state raises questions about security, particularly in multi-tenant settings. We plan to extend causal telemetry by adding a security model that incorporates authorization. This model will allow network owners to specify policies that control the flow of information, thereby ensuring that confidential information is not leaked via telemetry and also that telemetry data is not tainted.

8 RELATED WORK

Causal telemetry extends existing in-band telemetry systems [14, 16, 17, 32, 40]. These INT systems are popular despite their overheads, probably because they provide fine-grained visibility into network behavior. However, they either cannot relate behaviors occurring on different devices, or they rely on real-time clocks local to switches. So they lack mechanisms for tracking causality.

Speedlight [37] recognized the importance of causal consistency in the context of networks, adopting Lamport-Chandy snapshots for network executions. Compared to causal telemetry, Speedlight follows a sampling approach and constructs snapshots when directed by the network operator. As a result, Speedlight may miss important network events and cannot do post-hoc analyses. In contrast, causal graphs produced by CoCaTel can generate every consistent snapshot of a network execution.

Causal analysis on network logs [18, 22–24] recognized the importance of understanding the causal relationship between network events, and uses causal inference to mine the causality between network anomalies and their causes. Our work differs by using the happens-before relationship [25] whereas prior work by Jarry et al. [18] and Kobayashi et al. [22, 23, 24] relies on correlations between events. Both happens-before and correlation can be seen as approximations of "true" causality. Correlation fails to identify causally-related events that happen infrequently, whereas happens-before might deem a pair of events as related even when there is no causal connection between them. The false-positives that happens-before creates are easily dismissed by using further analysis. In contrast, the false-negatives generated by using correlations are not easily detected and, indeed, might hide the source of problematic behavior we are trying to analyze.

A number of prior systems have explored the design and implementation of systems for querying the state of a network [4, 8, 11, 15, 33, 39]. These systems have made advances in the design of high-level query languages and in the development of efficient compilation techniques, but the languages lack an expressive semantic foundation—particularly for queries involving network-wide state. Most prior work restricts the set of queries that can be posed to work around this limitation. For example, Sonata focuses on queries for a single switch [15], while Marple [32] focuses on queries that either (i) concern a single switch (ii) concerns a single packet or (iii) are associative and communicative so that the order of aggregation for sub-queries doesn't matter. We believe that causal graphs are a better foundation for network-wide queries.

The problem of constructing consistent global cuts and evaluating global predicates for distributed systems has been extensively studied in the distributed systems community [2, 5, 6, 30]. These techniques have been applied to distributed systems [3, 10, 29], where it is known as distributed tracing. We observe that modern networks have themselves become distributed systems [12], and hence propose adopting causal graphs [25] in the context of networks too.

ACKNOWLEDGMENTS

We are grateful to the EuroP4 reviewers for their feedback and suggestions for improving this paper. We also wish to thank Griffin Berlstein, who contributed many ideas to early discussions about causal telemetry, as well as our colleagues in the Cornell NetLab who gave generous and detailed feedback on this work. This work is supported by DARPA (HR001120C0107), NSF (1642120), and AFOSR (F9550-19-1-0264).

REFERENCES

[1] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication.* 503–514.
[2] Ozalp Babaoglu and Keith Marzullo. 1993. Consistent global states of distributed systems: Fundamental concepts and mechanisms. (1993).
[3] Victor Bahl, Paul Barham, Richard Black, Ranveer Chandra, Moises Goldszmidt, Rebecca Isaacs, Srikanth Kandula, Lun Li, John MacCormick, Dave Maltz, et al. 2006. Discovering dependencies for network management. (2006).
[4] Kevin Borders, Jonathan Springer, and Matthew Burnside. 2012. Chimera: A declarative language for streaming network traffic analysis. In *21st USENIX Security Symposium.* 365–379.
[5] K Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.
[6] Robert Cooper and Keith Marzullo. 1991. Consistent detection of global predicates. *ACM SIGPLAN Notices* 26, 12 (1991), 167–174.
[7] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.
[8] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. 2003. Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data.* 647–651.
[9] Antonin Bas et al. 2022. *Behavior Model Version 2.* Retrieved 2022-09-21 from https://github.com/p4lang/behavioral-model
[10] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. 2007. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*
[11] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A network programming language. *ACM Sigplan Notices* 46, 9 (2011), 279–291.
[12] Nate Foster, Nick McKeown, Jennifer Rexford, Guru Parulkar, Larry Peterson, and Oguz Sunay. 2020. Using deep programmability to put network owners in control. *ACM SIGCOMM Computer Communication Review* 50, 4 (2020), 82–88.
[13] IEEE 802.1 Working Group. 2008. *Priority-based Flow Control.* Retrieved 2022-09-21 from https://1.ieee802.org/dcb/802-1qbb/
[14] The P4.org Application Working Group. 2020. *In-band Network Telemetry (INT) Dataplane Specification.* Retrieved 2022-09-21 from https://p4.org/p4-spec/docs/INT_v2_1.pdf
[15] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication.* 357–371.
[16] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Maziéres, and Nick McKeown. 2012. Where is the debugger for my software-defined network?.

In *Proceedings of the first workshop on Hot topics in software defined networks*. 55–60.

[17] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 71–85.

[18] Richard Jarry, Satoru Kobayashi, and Kensuke Fukuda. 2021. A quantitative causal analysis for network log data. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 1437–1442.

[19] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proceedings of the Symposium on SDN Research*. Article 10.

[20] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 113–126.

[21] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 15–27.

[22] Satoru Kobayashi, Kazuki Otomo, and Kensuke Fukuda. 2019. Causal analysis of network logs with layered protocols and topology knowledge. In *2019 15th International Conference on Network and Service Management (CNSM)*. IEEE, 1–9.

[23] Satoru Kobayashi, Kazuki Otomo, Kensuke Fukuda, and Hiroshi Esaki. 2017. Mining causality of network events in log data. *IEEE Transactions on Network and Service Management* 15, 1 (2017), 53–67.

[24] Satoru Kobayashi, Keiichi Shima, Kenjiro Cho, Osamu Akashi, and Kensuke Fukuda. 2022. Comparative Causal Analysis of Network Log Data in Two Large ISPs. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–6.

[25] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)* 21, 7 (1978), 558–565.

[26] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. Flowradar: A better netflow for data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 311–324.

[27] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 101–114.

[28] Pedro Lopez, Juan-Miguel Martínez, and Jose Duato. 1998. A very efficient distributed deadlock detection mechanism for wormhole networks. In *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*. IEEE, 57–66.

[29] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 378–393.

[30] Keith Marzullo and Gil Neiger. 1991. Detection of global state predicates. In *International Workshop on Distributed Algorithms*. Springer, 254–272.

[31] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2014. Dream: dynamic resource allocation for software-defined measurement. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 419–430.

[32] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 85–98.

[33] Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker. 2016. Compiling path queries. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 207–222.

[34] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for Network Update. In *Proceedings of the 2012 ACM Conference on Special Interest Group on Data Communication*. 323–334.

[35] Alex Shpiner, Eitan Zahavi, Vladimir Zdornov, Tal Anker, and Matty Kadosh. 2016. Unlocking credit loop deadlocks. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. 85–91.

[36] Xinyu Crystal Wu and TS Eugene Ng. 2021. ITSY: Initial Trigger-Based PFC Deadlock Detection in the Data Plane. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 1–8.

[37] Nofel Yaseen, John Sonchack, and Vincent Liu. 2018. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 402–416.

[38] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 29–42.

[39] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. 2017. Quantitative network monitoring with netqre. In *Proceedings of the conference of the ACM special interest group on data communication*. 99–112.

[40] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. 2015. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 479–491.