The Case for an Internet Primitive for Fault Localization

William Sussman, Emily Marx, Venkat Arun, Akshay Narayan, Mohammad Alizadeh, Hari Balakrishnan, Aurojit Panda, Scott Shenker

MIT, UC Berkeley, NYU, ICSI

ABSTRACT

Modern distributed applications run across numerous microservices and components deployed in cloud datacenters, using shared cloud services for computing and storage, edge services such as content distribution networks, network functions such as rate limiters and firewalls, security infrastructures, network routers, and physical links. When a user-visible fault occurs, the first step toward diagnosis is *localization* to determine where the fault has occurred. However, because application delivery spans different layers and different organizations, no entity has complete visibility or access to the information required to localize faults quickly. This paper proposes a cross-layer, cross-domain, and cross-application fault localization primitive with a simple and standardized information interface for the Internet.

CCS CONCEPTS

• **Networks** → **Transport protocols**; *Network design principles*; *Programming interfaces*; **Cross-layer protocols**;

KEYWORDS

Fault Localization

ACM Reference Format:
William Sussman, Emily Marx, Venkat Arun, Akshay Narayan, Mohammad Alizadeh, Hari Balakrishnan, Aurojit Panda, Scott Shenker. 2022. The Case for an Internet Primitive for Fault Localization. In *The 20th ACM Workshop on Hot Topics in Networks (HotNets '22), November 14–15, 2022, Austin, TX, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3563766.3564105

1 INTRODUCTION

Since the advent of the web (thirty years ago) and mobile platforms (fifteen years ago), Internet applications have forever changed the way we live, work, and play. People now spend many hours everyday glued to their screens, so users are keenly aware of, and highly sensitive to, glitches, faults, and degraded app performance.

In response, application developers and operators have instrumented their software to track performance issues and faults [6]. Despite widespread instrumentation, logging, and tracing in today's software infrastructures, when a problem arises, often neither the user nor the application operator can identify what or where the problem is. Haven't we all been in videoconferences where someone freezes and everyone wonders if it's something at their end or elsewhere? These problems will become even more acute in the future with the promised proliferation of cyber-physical systems, networked robotics, augmented reality, and more.

When a user sees a problem, their first instinct is to quickly try to uncover what is wrong. The same is true when a component of an application detects that something is amiss from its monitoring: the first step to fixing the problem is uncovering *where* the misbehavior is. That is the problem we tackle in this paper: how to *quickly localize* a fault when it manifests itself visibly.

Despite years, indeed decades, of work on detecting and masking faults in distributed systems and applications (see §5), fault localization for modern Internet applications remains unsolved. Even when a simple fault such as a Wi-Fi connectivity issue occurs, users are stymied, and even well-resourced companies with experienced engineers don't know why user experience has degraded.

Why has it been difficult to identify which component is faulty when a problem is observed? Our belief is that it is because of the sheer number of things that must work properly for an application to function well. Today's large-scale applications are made up of many thousands of software components often running as microservices. These microservices run in virtual machines in cloud datacenters and use the physical network through virtual switches and network functions. Application components may be distributed across different datacenters globally and across cloud providers, and use components running in content distribution networks (CDNs) at edge locations and on client endpoints. Errors in any of these components, or in the myriad middleboxes, firewalls, network routers, and links that connect them, can lead to user-visible faults.

To locate an observed fault amidst this complexity, we need a mechanism that is:

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).

HotNets '22, November 14–15, 2022, Austin, TX, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9899-2/22/11…$15.00
https://doi.org/10.1145/3563766.3564105

- *cross-layer*, because a fault could occur anywhere from the physical layer to an app-layer library,
- *cross-domain*, because most applications use services across different organizations, and
- *cross-application*, because many components (*e.g.,* the network, middleboxes, cloud services, *etc.*) are shared by many applications.

This brings us to our position statement: *The Internet needs a universal fault localization primitive that is cross-layer, cross-domain, and cross-application.* We propose such a mechanism in this paper, which we call *WTF*.[1] Our goal is to stir up a debate about the value and practicality of creating a succinct information interface to localize faults affecting networked applications.

2 BACKGROUND

We start with some definitions. An *element* is any component, system service, network function, or device that can affect the observed behavior of a distributed application. A *node* is a server or virtual machine (VM) where one or more elements run. An *application problem* is a user-visible fault and an *element error* is any anomalous behavior (*e.g.,* crashes, overloads, misconfigurations, *etc.*) of an element. We focus on application problems resulting from one or more element errors and ignore other sources of application problems such as incorrect user inputs.

2.1 Localization Today

Trace analysis [6] is today a frequently used technique for localizing problems in distributed applications. It requires applications to record causal traces when processing requests, for which developers insert tracing library calls into their application components. Developers can use tools like Dapper [7], Zipkin [23], and Jaeger [11] to analyze these traces and identify faulty components. However, these traces only provide application- and library-level (e.g., gRPC [9], Thrift [26], *etc.*) information and do not include any information about platform services or the network.

Recent work[4, 12, 18, 22] develops techniques for collecting and analyzing traces from switches and other network components. However, these network trace collection and analysis tools do not provide sufficient information to localize many types of faults. There are two reasons why. First, these tools work at a single layer and cannot localize faults across layers, as evidenced by two recent GitHub incidents [13, 27]. Second, application-level trace analysis tools require complete logs from, and semantic knowledge about, all application-level elements. This assumption is impractical when using shared services managed by a cloud or edge provider. While one work in network tracing [4] considers elements (routers) across domains, it assumes that all routers implement identical semantics.

2.2 Example

We use an example to illustrate the challenges of cross-layer fault localization. We consider a multiplayer game application with two players (Player 1 and Player 2) streaming their game play to audience members. Player 1 is running the game on an instance provided by a cloud gaming service (*e.g.,* Google's Stadia, Microsoft's Xbox Cloud Gaming, or Nvidia's GeForce Now), while Player 2 runs the game locally. Both players are connected to a multiplayer game server. Player 1's cloud gaming instance is also connected to a streaming server, which transcodes game video and audio for audience members.

Because many games require quick reactions from players, high latencies cause application problems. For example, League of Legends, a popular multiplayer game, recently ran into trouble because of a 20 ms latency difference between two players [24]. Players might report a problem if there is a noticeable lag between a player performing an action and its effect becoming visible to other players. Audience members might report a problem if the video stream lags or has low quality.

Such problems can be caused by faults at either player's computer or network, Player 1's cloud gaming instance, the streaming server, or any of the routers and switches that interconnect these nodes. Furthermore, the problem at a node can be due to any of the elements running on the node, including application code (*e.g.,* the game, the game client, or video streaming service), GPU drivers of the players, audience members or gaming service, the hypervisor used by the cloud provider, and network stack at any node.

Consider a problem where an audience member reports a problem because the video and audio are out of sync. Let us consider how different entities might localize the problem. The streaming service provider can check if the problem is caused by a fault in the video transcoding service or due to problems with the stream received from the cloud gaming service. If the problem is with the received stream, then the streaming provider can do little more than inform the user who must then ask the cloud gaming service. Here, the cloud gaming service can only check if their streaming service is working correctly, and if it is they must turn to the game developer or the several network providers involved to localize the fault. The game developer can only localize problems in the game server (probably by trawling through massive logs) while each network provider can at best localize problems caused by their own links, switches and routers. Therefore, diagnosing this seemingly simple problem requires participation from many entities, and for the user to coordinate amongst them. Note also that the user does not have a commercial relationship with every entity in the process.

The bottom line is that no single entity has complete visibility into all the elements that might have caused the fault.

[1] Where's The Fault? (What did *you* think it stood for?!)

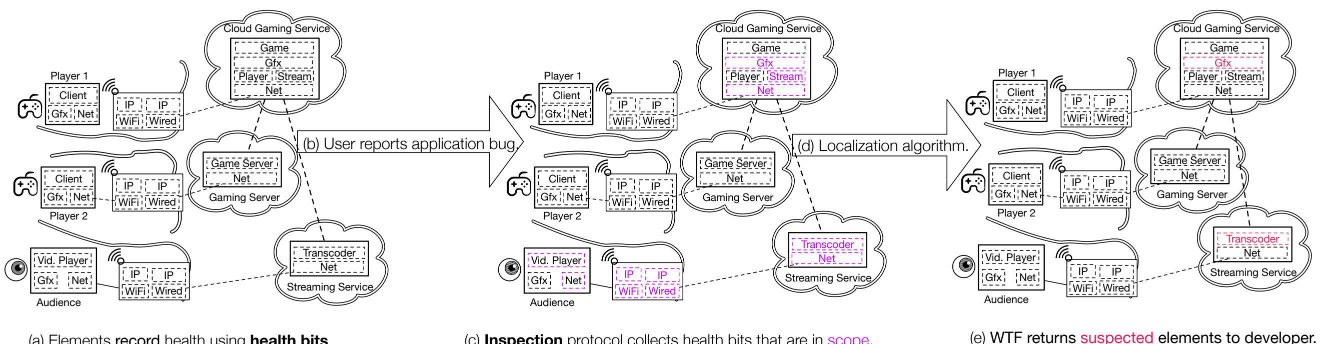

Figure 1: *WTF overview and example.*

3 WTF OVERVIEW

WTF is our proposed cross-layer, cross-domain, and cross-application fault localization mechanism. The key idea is for each element to maintain and expose a *succinct summary* of its state to aid fault localization. WTF does not focus on after-the-fact forensics or fault debugging once its location is identified.

3.1 Mechanisms

Our approach (Figure 1) requires that elements periodically log a small amount of information, called *health bits*, about their level of functioning. Health bits are a uniform and simple representation of element status (errors) encoded in a small number of bits (§4.1). They are tiny in size compared to the full context of the element's operational state to reduce the amount of data we need to analyze when localizing a problem.

The localization process collects health bits from elements likely to have contributed to a user-visible problem and infers the likely location of a fault. The key for making this approach general—across applications, layers, and domains—is for each element to determine its own health information in any way it wants, but to use a standard WTF-defined interface to provide this information to other interested elements and entities. WTF specifies the form of health bits, but does not specify how they should be set.

WTF has three mechanisms:

(1) A method that each element uses to periodically compute and store health bits and a protocol to expose its health bits to other entities using scoping rules under its control (§4.1).
(2) A protocol to inspect health bits across one or more elements to localize a problem (§4.2).
(3) A fault-localization algorithm that analyzes health bits across the elements when a user or app component notices a problem (§4.3).

In combination, these mechanisms are cross-layer, cross-domain, and cross-application, making WTF quite general in its applicability. However, WTF cannot localize all faults. As we discuss in §3.2, we cannot (and do not) prescribe what constitutes an element error, nor how an element sets its health bits. Thus, WTF can only localize problems that a user notices caused by faults that an element tracks. For instance, WTF cannot localize errors resulting from incorrect assumptions made by one element about the semantics of an RPC call made to another element. WTF is not designed to work in the face of Byzantine faults. Furthermore, WTF may not always precisely identify the root cause of a problem and instead may return a small subset of potentially faulty elements. Finally, because WTF assumes that elements record health bits for a bounded time, WTF cannot be used to localize faults that happened in the distant past.

Elements periodically maintain information about their health (*i.e.,* whether or not they have experienced any element errors since their last report) and on their perceptions about the health of other elements they interact with (*e.g.,* whether they received an unexpected return value or did not receive a response) using health bits. The decision of what constitutes an element error and the reporting period are up to the element's developers and administrators.

WTF allows users, automated failure detectors, or other entities to initiate localization (§4.3). In response, WTF runs a fault-localization algorithm over the health bit histories of the relevant elements used by the application. The main challenge lies in identifying the subset of elements that might have contributed to the problem whose health bits should be inspected. We refer to the set of health bits that are relevant to an application problem as the application problem's *scope*. Precisely identifying whether or not an element's health bits are within the scope of a problem requires additional context (*e.g.,* it might require a load balancer to record past decisions) and can add to the size of the element's trace (*i.e.,* their recent log of health bits along with the necessary context to help the localization know if these bits are relevant).

WTF is designed to work on a variety of nodes including network routers, where reducing trace size is desirable. On the other hand, limiting the problems for which an element's health bits are in scope is necessary to minimize inspection overheads, and to limit the amount of internal information revealed by an administrative domain. WTF's inspection protocol may include elements not on the path of the fault.

Finally, WTF passes the health bits collected to a localization algorithm that then produces its output. While we discuss how such an algorithm might work, our focus in this paper is on defining health bits and the inspection protocol. It is possible to develop several localization algorithms.

3.2 Design Principles and Implications

WTF is intentionally under-specified: it does not define what constitutes an element error, how often an element should generate health bits, or within what context (*e.g.,* who the element interacted with) the bits were generated. This lack of specificity is not a matter of choice. While we would like elements to report errors if and only if they caused application problems, this is impossible to know in advance because whether or not anomalous behavior at an element leads to an application problem depends on the application's semantics. For example, an increase in network delay has no user-visible impact on a text-messaging app, but can lead to user-visible problems in a videoconference. However, many elements, including the network, are shared between multiple applications and cannot make assumptions about the applications that use them. Thus, WTF can neither provide nor assume a precise specification for what an element's health bits denote.

WTF cannot specify the duration of time for which an element's health bits are stored, nor how much context is included, since the cost of storage varies significantly across node types and organizations. For example, most routers have limited memory and storage capacity and cannot store long histories, and network providers, a majority of whose elements run on routers, might prefer storing short histories. On the other hand, most cloud provider services run on servers where storage is less expensive and storing long sequences of health bits is cost-effective.

Our inability to tightly specify the emission of health bits might seem unusual and might appear to run counter to how systems are built today. We believe, however, that a loosely-specified, universal mechanism to report status (health bits) can be useful in practice because operators will develop conventions and configure elements so that the bits are set proportionate to the element's health (*e.g.,* a 2-bit specification can go from "all good" to "fatal" with two levels in between). This approach is similar to many successful Internet protocols including IP and BGP, which include fields with understated semantics and whose use has evolved and is governed by conventions developed from practical experience. For example, while the 1-bit explicit congestion notification (ECN) mark in packet headers is standardized, each router can determine if it is congested using its own algorithm; *i.e.,* the bit is standardized, but the logic used to set it is not. This method works because routers need only agree that a set ECN bit means that "there is congestion". Similarly, we believe that health bits can be used to indicate "directionally accurate" information about an element's status without standardizing or even revealing how the bits are set.

4 DESIGN

WTF's key design principle is that an element emits health bits based on locally-observable information about itself and its interactions with neighboring elements (a neighboring element may be at a higher, lower, or peer layer). WTF collects these health bits when an end-to-end fault is detected. This triggers the system to identify a set of elements that could have indicated the fault. We discuss three design challenges in making WTF practical: how elements might choose to emit bits (§4.1), how WTF can scope problems to a set of health bits (§4.2), and how fault localization algorithms might analyze health bit histories (§4.3).

4.1 Recording Element Health

To discuss how elements should emit health bits, we return to the example in §2.2. Here, the videogame program developer can output health bits regarding the multiplayer game's state (*e.g.,* by discretizing the players' latency to the server into health bits); the game streaming developer can output health bits regarding contention for cloud resources; Internet domains can output health bits regarding congestion; and end-users' home networks can output health bits regarding local network quality. Note that the availability of health bits from lower layers does not preclude setting health bits at higher layers. For example, if the game streaming developer implements an end-to-end latency check, this would still be useful information even if the same information is available at lower layers. This redundant information can help localize errors if health bits at different layers disagree on the health of the system.

A useful heuristic is that an element should emit health bits whenever a significant state change occurs; *e.g.,* a network switch might emit health bits when its queue length passes a configured threshold and when the queue length drops below a threshold, or perhaps whenever it sets the ECN bit. Further, an element should also emit health bits about its observations of its neighbor elements. This is important because an element may not itself be aware that it is responsible for a fault. For example, an element could be using an abnormally high amount of CPU cycles without knowing it, and the operating system of the node (itself an element) could emit bits indicating the element's unexpected behavior.

The element developer must set health bits roughly proportional to the severity of any error it might be experiencing. The element developer decides both this severity value and also when to emit health bits with the goal to produce them in a way that would most help subsequent localization. Service and domain administrators must decide how much health bit history they are willing to store depending on the how the elements in their domain tend to emit bits.

WTF's design must account for the possibility that the mechanism to collect or emit health bits could itself fail. Thus, we require that elements replicate their health bit output to their neighbors for fault tolerance. Alternately, a domain

could provide a logically centralized and fault-tolerant database to store health bit histories for all elements in that domain. While an element may not know its neighbors, it can attempt to discover them locally. One possible discovery mechanism is IP anycast, with which multiple nodes advertise the same IP address and the network forwards packets destined to the closest node with that address. A domain supporting WTF would forward a packet for this anycast IP to a node that can store health bits in a fault-tolerant way. If the element is in a domain that does not support WTF, the health bits would instead be delivered to the nearest domain with WTF support.

WTF's health bit emissions also provide an additional mechanism for detecting problems and initiating localization. When an element that is responsible for replicating a neighbor's health bits does not receive data to be replicated for a bit, it can deduce that either its neighbor has failed or is unable to communicate with it. For some applications, *e.g.,* email (where users may not directly observe problems for some time after they occur) using such a mechanism is necessary to ensure that localization is initiated before trace data related to the problem has been lost.

4.2 Scoping Health Bits

When an application problem occurs, WTF must determine the problem's scope. Recall that the scope is the set of health bits that are relevant to the problem. Thus, WTF must determine the set of elements that could have caused the application problem. Because this can be difficult to determine, we allow for false positives, which cause the returned scope to be larger than the true scope. Of course, returning too large a scope would add noise and frustrate the localization algorithm's efforts.

WTF's scoping mechanism should additionally be:

- *Decentralized*, so that domains can control what information to share with WTF.
- *Incrementally deployable and fault tolerant*, so that WTF can collect relevant health bits even if only a small fraction of elements participate, or similarly if many elements fail and are unable to return health bits.
- *Efficient*, with manageable network and storage requirements.

To perform this task, we envision WTF will be most useful in conjunction with a tracing mechanism, though we don't mandate the use of tracing for WTF to be usable. The element that notices the application problem will trigger a tracing mechanism and analyze the resulting scope (§4.3). The tracing mechanism recursively queries neighboring elements, nodes, and domains to gather their health bit histories. To allow nodes to compute scope, this request contains identifying application information such as IP addresses, geographic location, application name, and other application-level client identifiers.[2] Domains could record some of these features with each health bit and match on them when the query comes. One such example matching rule might be "match if the IP address falls within this subnet, or the app-level client identifier matches". The ability to use identifiers from different layers allows elements at any layer and any amount of visibility to participate in the protocol. As a last resort, elements can add a health bit that matches with all queries and include it in all scope queries, but this risks spuriously expanding the scope.

We note that this mechanism is opt-in. Domains can, for example, obscure details about their internal structure but still participate in WTF by responding to scoping queries with health bits for the domain overall (perhaps by issuing a domain-internal WTF query), while suppressing health bits from the component elements.

4.3 Using Health Bits to Localize Faults

Once an element has obtained the health bits relevant to a given application problem, how should it go about localizing the fault? Here we propose key considerations for a localization algorithm, leaving the development of specific algorithms and their evaluation to future work:

(1) *Severity*: an emitted health bit that indicates a higher severity is more *suspicious* (*i.e.,* more likely to have caused the fault) than one with a lower severity level.
(2) *Recency*: a more recently emitted health bit is more suspicious than an older one.
(3) *Comparison to counterfactual*: if an element's health bit history diverges from its history in cases where no fault is known to have occurred, we consider it to be more suspicious.

Of course, heuristics are fallible; there can be cases where they falsely indicate an element is suspicious. For this reason, we cannot prioritize the use of any one heuristic over another. Instead, we believe that approaches that combine the results of multiple heuristics (*e.g.,* approaches where we pick the union of all elements suspected by a set of heuristics or where we only pick elements suspected by a majority) will yield better localization.

Additionally, elements and applications can define their own localization functions which take the element-wise sequence of health bits as input. This might be useful if elements can utilize domain-specific knowledge about their application's topology of elements to implement additional heuristics. For example, if a given element is known to be a critical element of the application, a localization algorithm could check that element for indications of faults before any others.

5 RELATED WORK

Fault localization in distributed systems has been studied extensively. However, these prior approaches have not aimed

[2]To avoid the need for universal agreement, features should match only on application characteristics knowable to all elements.

to be cross-layer, cross-domain, or cross-application. They instead aim to localize problems within a single layer (often application code), require application integration, and have much more semantic information about the application.

Distributed Tracing. Many distributed applications collect request traces and use these traces for bug localization. These applications rely on tools such as Dapper [7], Zipkin [23], Jaeger [11], and OpenTelemetry [3] to collect request traces. These tools extend techniques proposed by X-Trace [6] and subsequent projects [5, 14, 16, 17, 19–21, 25]. They log a causally-ordered trace by following a request through application components and provide tools to centrally aggregate, analyze, and localize bugs from these logs.

Internet-scale Debugging. Windows Error Reporting [8] uses error statistics to localize cross-application bugs that were otherwise invisible at small scales. BlameIt [12], Secure Packet Provenance [4], and Packet Obituaries [1] identify and localize faults across network domains. BlameIt relies on measurements from hosts as well as active probing to localize instances of high latency to "cloud, middle, or client". Secure Packet Provenance proposes a cross-domain header protocol used to share and collect telemetry between different networks. Finally, the Packet Obituaries proposal sought to provide hosts with information about where their packets were dropped.

Network Monitoring. NetPoirot [2], Pingmesh [10], and TRat [30] perform network monitoring and debugging tasks using information from end-hosts. Marple [22], UnivMon [18], and OpenSketch [29] provide network developers with increased visibility into their networks by taking advantage of advances in programmable switches. These systems are promising sources to draw from when setting health bits.

Cross-layer Insights. A few systems have attempted to combine insights from multiple layers to localize application problems. WhyHigh [15] used active probes as well as BGP information to diagnose latency spikes. More recently, Sage [28] uses machine learning to correlate low-level system performance information with problems in microservices within a datacenter. This approach indicates that localization algorithms will be able to similarly draw useful conclusions from health bits.

6 DISCUSSION AND CONCLUSION

WTF proposes a non-traditional systems approach (but a traditional Internet approach) for building a fault localization mechanism for large-scale applications. Conventional wisdom dictates that to be useful, a system must be precisely specified, *i.e.,* the developer must carefully define its inputs and assumptions, and use algorithms that guarantee correctness as long as all inputs are correctly provided and all assumptions hold. Our work builds on the observation that well-specified systems are hard, if not impossible, to deploy in environments made up of components managed and built by multiple independent entities. This has been the case for the Internet, where coordinating protocol deployments (through flag days or other means) has been impossible for decades, and therefore systems have been designed to rely on loose specifications and assume nothing about how the information between components is computed. Because networked applications often span multiple domains and layers, and are built from many thousands of components, they resemble smaller versions of the Internet; thus, the philosophy underlying the design and deployment of Internet components may provide useful insights for fault localization.

We conclude with a brief discussion of some open questions for WTF.

Incentives. One concern for WTF's deployment prospects is that while WTF itself is merely a system for collecting and analyzing health-bits, element developers might prefer to avoid sharing the health status of their systems to avoid being assigned culpability for faults they report. However, we note that the reverse is equally true; WTF can help element developers avoid spurious claims of faults when their element does not malfunction. Further, because of our choice of a loosely-defined health-bit specification, operators can also choose the granularity of their health-bits to mitigate privacy or security concerns. For example, domain administrators who are unwilling to reveal their network topologies can aggregate health-bits across multiple nodes in the domain so they appear to be emitted by a single node.

Incremental Deployment. While localization is most useful when all elements involved in a service (including network elements that connect the various participants) support WTF, our proposal is beneficial even when only a subset of elements report health-bits. For example, consider a microservice application written by multiple teams within a single enterprise. This enterprise could adopt WTF to localize faults within the microservice application to reduce the amount of cross-team coordination required during debugging. Further, even in environments with elements provided by fully uncoordinated elements, health-bits from the subset of elements that report them narrow the scope of fault localization queries.

We believe WTF's adoption will be driven by demand from users and application developers/operators; currently, when application problems emerge, neither users nor application operators can localize the problem. For users, this means that they do not know who to report problems to and thus cannot quickly resolve them, while for developers this leads to low user satisfaction and falling revenues [12].

Overall, we believe that by adopting the concept of health-bits and specifying very little, WTF provides a design that is both useful for applications that span domains and layers, and is deployable incrementally.

Acknowledgements. This work is supported by NSF grants 1817115 and 2145471. We thank Wen Zhang for comments on drafts of this paper.

REFERENCES

[1] Katerina Argyraki, Petros Maniatis, David Cheriton, and Scott Shenker. 2004. Providing Packet Obituaries. In *HotNets*. 5

[2] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. 2016. Taking the Blame Game out of Data Centers Operations with NetPoirot. In *SIGCOMM*. https://doi.org/10.1145/2934872.2934884 5

[3] The OpenTelemetry Authors. 2022. OpenTelemetry. https://opentelemetry.io/. (2022). 5

[4] Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2017. One Primitive to Diagnose Them All: Architectural Support for Internet Diagnostics. In *EuroSys*. https://doi.org/10.1145/3064176.3064212 2.1, 5

[5] Rodrigo Fonseca and Jonathan Mace. 2015. We are Losing Track: A Case for Causal Metadata in Distributed Systems. In *HPTS*. 5

[6] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*. 1, 2.1, 5

[7] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. 2017. Dapper: Data Plane Performance Diagnosis of TCP. In *SoSR*. https://doi.org/10.1145/3050220.3050228 2.1, 5

[8] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *SOSP*. https://doi.org/10.1145/1629575.1629586 5

[9] Google. 2022. gRPC: A high performance, open-source, universal RPC framework. https://grpc.io. (2022). 2.1

[10] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *SIGCOMM*. https://doi.org/10.1145/2785956.2787496 5

[11] jaeger 2022. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/. (2022). 2.1, 5

[12] Yuchen Jin, Sundararajan Renganathan, Ganesh Ananthanarayanan, Junchen Jiang, Venkata N. Padmanabhan, Manuel Schroder, Matt Calder, and Arvind Krishnamurthy. 2019. Zooming in on Wide-Area Latencies to a Global Cloud Provider. In *SIGCOMM*. https://doi.org/10.1145/3341302.3342073 2.1, 5, 6

[13] Theo Julienne. 2019. Debugging network stalls on Kubernetes. https://github.blog/2019-11-21-debugging-network-stalls-on-kubernetes/. (2019). 2.1

[14] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *SOSP*. https://doi.org/10.1145/3132747.3132749 5

[15] Rupa Krishnan, Harsha V. Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas Anderson, and Jie Gao. 2009. Moving beyond End-to-End Path Information to Optimize CDN Performance. In *IMC*. https://doi.org/10.1145/1644893.1644917 5

[16] Pedro Las-Casas, Jonathan Mace, Dorgival Guedes, and Rodrigo Fonseca. 2018. Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay. In *SoCC*. https://doi.org/10.1145/3267809.3267841 5

[17] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. 2019. Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering. In *SoCC*. https://doi.org/10.1145/3357223.3362736 5

[18] 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 *SIGCOMM*. https://doi.org/10.1145/2934872.2934906 2.1, 5

[19] Jonathan Mace. 2018. *A Universal Architecture for Cross-Cutting Tools in Distributed Systems*. Ph.D. Dissertation. Brown University. 5

[20] Jonathan Mace and Rodrigo Fonseca. 2018. Universal Context Propagation for Distributed System Instrumentation. In *EuroSys*. https://doi.org/10.1145/3190508.3190526

[21] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *SOSP*. https://doi.org/10.1145/2815400.2815415 5

[22] 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 *SIGCOMM*. https://doi.org/10.1145/3098822.3098829 2.1, 5

[23] OpenZipkin. 2022. OpenZipkin: A Distributed Tracing System. https://zipkin.io/. (2022). 2.1, 5

[24] Riot Games. 2022. Riot Games Tech Blog: Artificial Latency for Remote Competitors. https://lolesports.com/article/riot-games-tech-blog-artificial-latency-for-remote-competitors/blt44154a33b5d5a616. (2022). 2.2

[25] Raja R. Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H. Sigelman, Rodrigo Fonseca, and Gregory R. Ganger. 2016. Principled Workflow-Centric Tracing of Distributed Systems. In *SoCC*. https://doi.org/10.1145/2987550.2987568 5

[26] Thrift [n. d.]. Apache Thrift. https://thrift.apache.org/. ([n. d.]). 2.1

[27] Jason Warner. 2018. October 21 post-incident analysis. https://github.blog/2018-10-30-oct21-post-incident-analysis/. (2018). 2.1

[28] Jane Yen, Tamás Lévai, Qinyuan Ye, Xiang Ren, Ramesh Govindan, and Barath Raghavan. 2021. Semi-Automated Protocol Disambiguation and Code Generation. In *SIGCOMM*. https://doi.org/10.1145/3452296.3472910 5

[29] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *NSDI*. 5

[30] Yin Zhang, Lee Breslau, Vern Paxson, and Scott Shenker. 2002. On the Characteristics and Origins of Internet Flow Rates. *SIGCOMM CCR* 32, 4 (aug 2002). https://doi.org/10.1145/964725.633055 5