Jonathan Mace - Research Statement

November 2017

Distributed systems represent some of the most interesting and successful computing applications in use today, from web search and social networks, to data analytics and large-scale machine learning, to loosely-coupled microservices, serverless lambdas, and the public cloud. However, it is notoriously difficult to understand, troubleshoot, and enforce distributed systems behaviors, because unlike standalone programs they lack a central point of visibility and control: each component only performs a narrow slice of work, and to execute a high-level task (such as a search query or an analytics job) entails a complex flow across multiple processes, machines, and the network. When problems occur in a distributed system, their symptoms are often far removed from the root cause, transient factors are often to blame, and instead of an outright crash, systems experience more pernicious symptoms like sustained degraded performance. When they materialize in production systems (as they have with all major cloud providers) it leads to high-profile outages and a massive loss of revenue.

My research focuses on improving our ability to observe, reason about, and enforce end-to-end behaviors in distributed systems. The main approach I have taken is to develop *cross-cutting tools* that co-ordinate across system components and layers at runtime, re-establishing end-to-end visibility and control over different system behaviors. My work includes tools for co-ordinating scheduler parameters across shared systems to give users real-time performance guarantees (**NSDI '15**) [9]; for monitoring metrics and properties to answer ad-hoc queries from users at runtime (**SOSP '15, Best Paper Award**) [12]; and for recording structured performance traces and analyzing them in aggregate (**SOSP '17**) [2]. A recurring theme of my work is to seek a global understanding of complex software stacks, peering through abstractions and layers to observe, and exploit, the end-to-end flow of execution. In practical terms this ranges from the underlying mechanisms for communicating information and orchestrating actions across component boundaries, to high-level abstractions that align the user's perspective with the cross-cutting task, hiding system-level communication and co-ordination concerns.

Throughout my research I have been motivated by real problems experienced by developers and operators in practice, and one of my primary goals is to design and implement practical tools that can feasibly be deployed in production systems. This is especially delicate for cross-cutting tasks, which are difficult to establish in practice because of amplified logistical costs – they require coherent developer choices and pervasive deployment across all inter-operating systems and components [11]. This leads me to gravitate towards simple, elegant solutions to these otherwise messy problems, and I seek abstractions that minimize the effort required to develop and deploy cross-cutting tools. I am also heavily involved with the distributed tracing workgroup, the main industry group for this topic, and I sit on the OpenTracing industrial advisory board. All of my research artifacts have been open-sourced; concepts introduced by my work have received mainstream adoption and are deployed in production at numerous companies. I will continue to collaborate with these groups, and other partners in industry, to identify challenges, validate solutions, and help advance the state-of-the-art

In what follows I will present four areas of my current research, which paint a broad picture of the themes and overarching challenges in performing cross-cutting distributed systems tasks. Finally, I will outline future directions for my research that I intend to pursue in the next stage of my research career.

Current Research

Real-Time End-To-End Resource Management

My initial thesis work focused on the management of resources – e.g. CPU, disk and network bandwidth, lock and threadpool usage – in multi-tenant distributed systems such as the Hadoop stack, where the workloads of different tenants execute within shared system processes. It is hard to apply global resource policies because there is no co-ordination between local mechanisms within each process [8].

To address this I built Retro (**NSDI'15**) [9] a framework for co-ordinated per-tenant resource policies. Retro measures the resources used by each tenant across all components, then co-ordinates local schedulers on a per-tenant granularity, to achieve global policies. For example, if tenant A is not getting their desired end-to-end latency, Retro can (i) find out which resource is to blame (e.g., an overutilized disk); (ii) find which other tenants are using that resource; and (iii) throttle only those offending tenants to restore tenant A's latency. Retro makes these decisions in a *logically centralized controller*, similar in spirit to software defined networking. The centralized controller receives incoming resource measurements in real-time to paint a global picture of each tenant's resource demands. With this visibility, a policy can converge towards its goal by continually reallocating resources to tenants and adjusting scheduler weights across the system.

A key design principle of Retro is to separate policies from the mechanisms to enforce them. Retro introduces abstractions for reasoning about resources and tenants that make it easy to write concise resource policies that run on the controller; I demonstrated Retro with policies for bottleneck fairness, latency guarantees, and dominant-resource fairness, each in less than 20 lines of code. Policies are easy to understand and maintain, and the separation allows them to be reused across different systems or extended with more resources.

In general, Retro's design is well suited for cross-cutting tasks that are coarse-grained and long-lived. For example, Retro converges to *aggregate* workload goals for each tenant over time; this permits an out-of-band feedback loop with a ~1 second delay between reporting measurements and updating scheduler parameters. At the system level, Retro's mechanisms generalize more broadly. To attribute measurements and schedule requests per-tenant, Retro propagates a tenant ID alongside requests and across system components; this establishes a tenant context that otherwise does not exist. In general, propagating metadata between tool invocations is a pattern that applies to many cross-cutting tasks, and one that I revisit in my later work.

Monitoring and Correlating Metrics on the Fly

Retro achieves end-to-end visibility by including tenant IDs with executions as they traverse system components. My subsequent work on Pivot Tracing (**SOSP'15, Best Paper Award**) [12] extends this approach to monitoring and correlating arbitrary metrics at runtime. Pivot Tracing generalizes the metadata propagation of previous causal tracing work (e.g. Retro's tenant ID) and combines it with dynamic instrumentation of running systems. Users issue SQL-like queries about properties of the system, to collect arbitrary metrics from one point, while being able to select, filter, and group by concepts meaningful at other parts of the system. For example, a possible query could count database accesses across all processes, grouped by the 'web user' — even if the 'web user' is only known briefly while the request passes through the front end, and the database is only accessed later after several indirect calls through backend services. Using Pivot Tracing, we were able to interactively diagnose multiple obtuse bugs in the Hadoop stack that were not reflected in any existing monitoring.

Pivot Tracing's strength lies in its *happened-before join* query operator, which provides an intuitive way for users to relate inputs from multiple sources (including across component boundaries). For any two inputs (e.g. metrics, attributes, events), a happened-before join combines them if (i) they occur as part of the same execution (e.g. the same search request or analytics job); and (ii) the first input causally precedes the second. Pivot Tracing efficiently evaluates happened-before join *in-band* by carrying partial query results alongside execution from the first join input to the second, using a dynamic version of the metadata propagated in previous tracing work (e.g. Retro's tenant ID). Dynamic instrumentation lets users define and install queries into live systems on the fly, and means queries incur zero overhead until they are actually installed.

Pivot Tracing demonstrates a compelling approach to the design of cross-cutting tools. Dynamic instrumentation is useful for adding and removing monitoring and enforcement code on the fly. Dynamic metadata provides a flexible mechanism for passing information with the flow of execution, eschewing expensive and imprecise alternatives such as out-of-band joins or dependency inference. Pivot Tracing has attracted significant interest from the community and from industry; versions of the SOSP paper and talk were invited to appear at the USENIX ATC conference, in the USENIX ;login: magazine, in the Research Highlights of the Communications of the ACM, and in the ACM Transactions on Computer Systems journal.

Monitoring and Tracing Systems at Scale

Pivot Tracing enables ad-hoc monitoring of metrics and correlations, but it inherently does not support historical analysis. At large internet companies like Facebook, historical analysis is important for exploring trends and correlations in metrics over time. During my research collaboration with Facebook I helped to design and develop Canopy (**SOSP'17**) [2], a system for ad-hoc exploratory analysis of causally-related performance metrics across the entire Facebook software stack. Canopy's design addresses an important scalability question – how do you collect, store, and interactively query billions of performance traces? Today, Canopy is deployed in production at Facebook and collects over 1.1 billion traces per day.

The main entry point for Canopy users is through high-level *datasets*, in which each row represents one execution (e.g. a request to load facebook.com) and each column is a feature of the request, e.g.

Request ID	*Device*	*Network*	*Country*	*Latency*	*Net Bytes*	*CSS Bytes*	*DB Queries*	...
4af9b72f	Galaxy S3	4G LTE	USA	2356ms	300kB	80kB	8	...
01e1ae00	Pixel	3G	GB	4156ms	310kB	85kB	9	...

Canopy datasets comprise hundreds of features and billions of requests; they are stored in an in-memory time series database, enabling Facebook engineers to query and analyze aggregate performance data in real-time. Datasets derive from *end-to-end traces*, which capture performance data and the causal ordering of events in each execution. Canopy relates data across components by propagating 'breadcrumbs' with executions and attaching them to recorded data; this enables Canopy's backend to collect events relating to the same execution and reconstruct their causality. Canopy's backend then turns the data into a higher-level representation of an execution that is easier to reason about, and finally evaluates user-defined feature extraction functions that populate the output datasets. Canopy only materializes features specified a priori by Facebook engineers; traces are rich objects comprising thousands of events plus their causal relationships, so the number of potentially interesting features is enormous. To offset this, Canopy makes it is easy for engineers to specify and deploy new features, update datasets, and revisit old traces should the need arise; it is also easy to extend instrumentation and incorporate new performance data into future traces.

To develop Canopy we had to address several logistical issues in deploying a cross-cutting tool at scale, such as tolerating a wide range of data types and execution models, and incorporating data across heterogeneous systems, products, and developers. We made sure that engineers would have the right data representations, abstractions, and APIs for different tasks, from instrumenting systems, to extracting features from traces, to querying and visualizing custom datasets.

Canopy addresses the question of how to collect and expose end-to-end traces at large scale. Looking forward however, Canopy effectively provides an umbrella of visibility across different systems and performance data. It not only enables direct monitoring and analysis of end-to-end performance data, but it also opens the door to experimentation using statistics, data mining, and other techniques, to derive higher-level understanding of system behaviors. Canopy is a useful aid when designing other cross-cutting tools, e.g. new tools for online enforcement tasks like Retro, as it provides a way to explore potential data sources for the new tool.

Universal Abstractions for Tracing

A recurring theme with the tools I have described is that they propagate metadata alongside executions at runtime: Retro propagates a tenant ID; Pivot Tracing propagates partial query state; Canopy propagates breadcrumbs. In general, metadata propagation enables downstream components to observe, and react to, events that occurred previously during an execution. It is a compelling mechanism for any tool that wants to observe and enforce behaviors at the granularity of end-to-end executions [1]. However, instrumenting a system to propagate metadata is time consuming and labor intensive, as it entails modifications to source code that touch nearly every component of a distributed system.

My most recent work on *baggage contexts* [11] exploits a simple observation that cross-cutting tools all share *how* metadata is propagated with the execution, and only differ in *what* metadata gets propagated. Baggage is a general underlying format for cross-cutting metadata that enables instrumentation to be shared and reused by different tools, essentially making it a 'one-time' task. System components propagate baggage in an opaque way, oblivious to the semantics and metadata of the cross-cutting tools that may use it. The main challenge in doing this – which baggage solves – is how to maintain correctness under arbitrary concurrency; specifically, merging metadata when concurrent execution branches join, without interpreting the metadata's meaning. Baggage specifies a simple binary merge operation that is idempotent and order-preserving; these properties provide generality and enable efficient encodings for many data types, including primitives, IDs, sets, maps, counters, nested data structures, and several more based on conflict-free replicated data types. Baggage also enables the multiplexing of different tracing tools. I developed and open-sourced baggage libraries in several languages [5]; to demonstrate baggage, I implemented revised and extended versions of seven cross-cutting tools (including Retro and Pivot Tracing) [4] and deployed them side-by-side in several baggage-instrumented systems.

Baggage is a step forward towards truly pervasive instrumentation of distributed systems, addressing important roadblocks. At the system level, it increases the value of instrumenting a system as such instrumentation can be re-used by many tools. It also makes the work of tool developers much easier, as they can focus on tool logic and data types, and ignore details of serialization, deserialization, propagation, and all of the subtleties of keeping data consistent in face of concurrency. Baggage brings in all the standard benefits of a strong separation of concerns, reuse, and independent evolution around a simple yet expressive narrow waist. In industry, practitioners have seized upon the notion of baggage and incorporated it into popular open-source tracing frameworks like OpenTracing, Zipkin, and Jaeger. Baggage has enabled organizations to rapidly prototype new features; for example in its deployment over the past year at Uber, engineers have deployed tools for black-box probing, capacity planning, authorization and claims, cost accounting, chaos engineering, and contextualized metrics, that previously would have been infeasible.

Future Research

Cloud and distributed systems are still a rapidly changing area. I believe that the fundamental themes of my work will only grow in importance as the area matures, and I intend to continue exploring these topics in my future work. Looking further forward, my interests extend to how some of these ideas might be established in practice, and the common designs and conventions that might emerge over time.

Reusable Abstractions Deploying cross-cutting tools inherently requires coherent choices and participation across all system components. However, today there is little consensus on which tools, abstractions, and approaches to use. Developers of different components are often isolated from one another, causing them to make incompatible or conflicting choices about the cross-cutting tools to embed. Ideally it is not developers who should make these choices, but the operators who deploy the systems at runtime. I am interested in pursuing general-purpose mechanisms that developers can embed in the system at development time, to enable operators to dynamically deploy *any* cross-cutting tool at runtime. My initial work on baggage is a step towards this broader goal. A compelling way to explore this question is to take inspiration from software defined networking, and consider how cross-cutting tools can decouple system-wide enforcement mechanisms

from tool-specific control logic. By disentangling control logic from enforcement, we can expose and exploit commonalities in the way different tools observe and manipulate system behaviors.

Large-Scale Performance Analytics With systems like Canopy, we have effectively addressed many of the challenges of capturing rich end-to-end performance traces in large-scale systems. For the intrepid systems researcher, this presents a conundrum — we have been so focused on *how* to collect performance data from systems, that we don't know what to do with it once we have it. I am interested in deriving high-level understanding of the way systems behave by applying techniques from statistics, data mining, and machine learning to large volumes of performance traces. There are a wide range of potential use cases, and they extend beyond just analysis. For example, traces can be used to find features early in an execution that are highly predictive of later performance; this insight can then be exploited by cross-cutting tools at runtime to make better scheduling decisions. One of the key challenges of analyzing performance traces is incorporating their structure, as a trace is conceptually a directed, acyclic graph (DAG) of events, with annotations (e.g. labels and metrics) on events and on edges; the most interesting features are often the structural relationships between events, such as their ordering and timing. This structure makes many off-the-shelf techniques computationally intractable given the number of potential features; it demands new approaches to visualizing, querying, and exploring traces based on structure; and at scale, it imposes new storage and processing constraints.

Automatic Instrumentation Instrumentation has long been the biggest pain point of deploying tracing tools. My work on baggage reduces, but does not fully remove, the instrumentation burden associated with deploying cross-cutting tools. In the distributed tracing community, as well as the broader application performance monitoring (APM) industry, fully automatic instrumentation is a panacea: if the instrumentation burden can be completely removed, then it potentially enables black-box deployment of many cross-cutting tools. I believe that fully automatic instrumentation is possible, based on the observation that instrumentation is about capturing the flow of execution, and most programs adhere to a very small set of execution models — e.g. threads, queues, RPCs, etc. Instrumentation is challenging *not* because it is complex, but because it must be pervasive; most instrumentation is simple, repetitive, and characterized by only a handful of different techniques for capturing different models. In my future work, I intend to explore methods for observing or inferring execution models in systems, with the goal of reducing or eliminating the need for manual instrumentation. If successful, this would represent a significant advance for both the research community and industry.

Management Tools My work so far has explored several promising avenues for improving our visibility and control over distributed systems. However, as we discover new designs for distributed systems, new use cases, and new requirements, we will also encounter new and interesting dimensions along which they can fail. I will continue to explore cross-cutting tools for production systems, and I am also interested in revisiting tools and techniques that are well understood for standalone programs but may lack distributed systems analogues. For example, are there equivalents to stop the world or time-traveling debuggers that can work on a production distributed system?

References

[1] Rodrigo Fonseca and **Jonathan Mace**. We are Losing Track: a Case for Causal Metadata in Distributed Systems. In *Proceedings of the 15th International Workshop on High Performance Transaction Systems (HPTS)*, October 2015.

[2] Jonathan Kaldor, **Jonathan Mace**, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Vekataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, October 2017.

[3] Raja R Sambasivan, Ilari Shafer, **Jonathan Mace**, Benjamin H Sigelman, Rodrigo Fonseca, and Gregory R Ganger. Principled Workflow-Centric Tracing of Distributed Systems. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, October 2016.

[4] **Jonathan Mace**. Brown University Tracing Framework. Retrieved November 2017 from `https://github.com/brownsys/tracing-framework`.

[5] **Jonathan Mace**. The Tracing Plane GitHub Repository. Retrieved November 2017 from `https://github.com/tracingplane`.

[6] **Jonathan Mace**. Revisiting End-to-End Trace Comparison with Graph Kernels. M.Sc. Project, Brown University, 2013.

[7] **Jonathan Mace**. End-to-End Tracing: Adoption and Use Cases. Survey, Brown University, 2017.

[8] **Jonathan Mace**, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Towards General-Purpose Resource Management in Shared Cloud Services. In *Proceedings of the 10th Workshop on Hot Topics in System Dependability (HotDep)*, October 2014.

[9] **Jonathan Mace**, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2015.

[10] **Jonathan Mace**, Peter Bodik, Madanlal Musuvathi, Rodrigo Fonseca, and Krishnan Varadarajan. 2DFQ: Two-Dimensional Fair Queuing for Multi-Tenant Cloud Services. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM)*, August 2016.

[11] **Jonathan Mace** and Rodrigo Fonseca. Universal Context Propagation for Distributed System Instrumentation. *In Submission*.

[12] **Jonathan Mace**, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, October 2015. ***Best Paper Award***. Also to appear in *Communications of the ACM (CACM)* and *ACM Transactions on Computer Systems (TOCS)*.