Experiences with Tracing Causality in Networked Services

Rodrigo Fonseca
Brown University

Michael J. Freedman
Princeton University

George Porter
UC San Diego

Abstract

Unlike device-centric monitoring, task-centric tracing enables an operator to causally trace the complete execution of a networked system across the boundaries of applications, protocols, and administrative domains. In this paper, we argue that causal, end-to-end tracing should be an integral part of network services. Moreover, it is not fundamentally difficult to achieve, given a primitive that propagates task metadata alongside logical execution and communication paths.

X-Trace is a framework that relies on such propagation to provide comprehensive causal tracing. We report on our experience integrating X-Trace into several production networked services—including 802.1X authentication, Web content distribution, and DNS-based replica selection—to illustrate benefits of causal tracing, and to discuss the instrumentation of different protocols and component architectures. We highlight the challenges we encountered and techniques we developed to better integrate causal tracing into network services.

1 Introduction

As the scale and complexity of networked services increases, so do the challenges of developing, deploying, managing, and troubleshooting them. These services routinely depend on equipment and software produced by a number of different parties. While standard protocols and interfaces typically govern their components' interactions, there is no standard way to integrate management, monitoring, or diagnostics. Common solutions to this problem are *device*-centric rather than end-to-end *task*-centric, and they rely on ad-hoc, brute-force log analysis [19] or inference techniques [13].

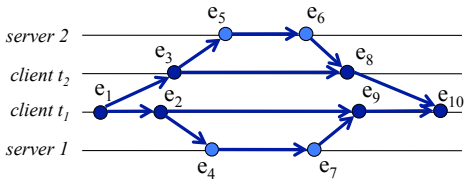
We argue that networked services should be built with end-to-end, task-centric monitoring as a first-class concept. It should be possible to follow a request through a networked service from start to finish, with variable level of detail, and across application and network layers. Such a mechanism should be incrementally deployable, and it should be composable by components provided by different parties, much like services themselves.

The alternative to such a model—the status quo—sacrifices visibility, causality, and completeness. Trying to reconstruct the causality of a task across components is generally done by joining logs on time and on application-specific, ad-hoc identifiers [4]. This is tedious and error-prone, relies on well-synchronized clocks, and requires extensive knowledge of the applications involved. Furthermore, it requires that all operations be logged at components to guarantee coverage. Causal end-to-end traces, on the other hand, sidestep these problems: they allow one to naturally join events on different nodes, and they can detect and even fix time-synchronization issues. They also allow a powerful task-based sampling of events: It becomes possible to record all information pertaining to a given task across nodes in great detail, while ignoring other extraneous events. This *coherent sampling* is of critical import at scale.

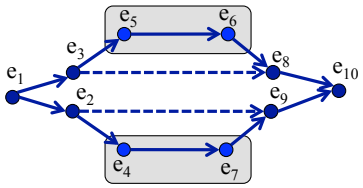
On the flip-side, both the end-to-end tracing mechanism we developed, X-Trace [10], and those of other projects with similar goals [6, 18], require that developers explicitly modify source code to carry end-to-end metadata throughout a task's computation and communication. A significant concern is that such a process would be infeasible, even if source code is available, given the complexity of real-world network systems.

This paper addresses this concern by reporting on the integration and experimentation with X-Trace in several production services: an 802.1X authentication infrastructure, CoralCDN [11], and the OASIS anycast service [12]. X-Trace' causal, end-to-end visibility into these services enabled the discovery of a number of bugs and the diagnosis of performance faults, which we describe. Together, these services cover both thread and event-based architectures, vary in maturity and code size, and consider both local and wide-area network systems.

In all these settings, we found that small additions to the systems' communication libraries (*e.g.*, RPC or HTTP) or concurrency management mechanisms (*e.g.*, thread pools, event hooks, or continuation passing) was sufficient for tracking end-to-end causality throughout the services. Subsequently, programmers typically only needed to add simple log statements for recording metadata. That said, there were several subtleties and challenges that arose across multiple systems. Thus, the final contribution of this paper are recommendations for how to better integrate end-to-end tracing into a variety of system architectures and implementations.



(a) Two parallel RPCs.



(b) Corresponding X-Trace representation.

Figure 1: **Example RPC calls and X-Trace representation. Edges e_3e_8 and e_2e_9 are *redundant*, abstracting the boxed subgraphs.**

2 X-Trace Overview

X-Trace is a framework that allows an operator to trace the execution of a networked system across the boundaries of applications, protocols, and administrative domains [9, 10]. X-Trace represents discrete *events* in an execution of a distributed system and their causal relation. Events are grouped into logical *tasks*. Tasks generally have an intuitive meaning in the context of the traced application. They have a well-defined starting event and comprise all of the causally-related subsequent events, *e.g.*, an HTTP request to a content distribution network (CDN), or a user request to a network authentication service (such as 802.1X).

Each task in X-Trace receives a probabilistically unique task id, and within a task each event receives a probabilistic unique event id. Events are created by X-Trace logging statements in source code. Each X-Trace event generates a report, which contains information about the even, and the id of zero or more preceding events. To accomplish this, X-Trace must carry constant-sized metadata along a process's computation and across communication boundaries; this metadata corresponds to the task and event ids of the last causally-related event. While this metadata is kept in-band, reports are sent to a collection service out-of-band. This practice both minimizes the overhead and decouples the fate of reports from the fate of the execution.

A system instrumented with X-Trace outputs a set of task graphs, directed acyclic graphs connecting events according to Lamport's *happens before* relation. X-Trace does not rely on clocks, but on the sequence of event execution. Because of the propagation of metadata among related events, X-Trace captures true causality, rather than incidental causality due to execution ordering.

Figure 1 shows an example graph for parallel RPCs between a client and two servers. The task graph is constructed offline by collecting reports for each event. A task graph captures the concurrency in the task's execution—and in fact, semantic causality—such as the two parallel sequence of events $e_1e_3e_5e_6e_8e_{10}$ and $e_1e_2e_4e_7e_9e_{10}$ in Figure 1. It also captures the abstraction of a subgraph by an edge. In the same figure, the edge between e_3 and e_8 abstracts, from the point-of-view of the client, the subgraph $\{e_5, e_6\}$. We call an edge such as e_3e_8 a redundant edge, and can determine the redundant edges when post-processing the task graph, by doing a transitive reduction of the graph. These edges are useful for summarizing the graph and highlighting the concurrent structure of the tasks.

X-Trace requires no prior coordination, as ids are probabilistically unique. It only requires carrying mutable and fixed-size metadata in protocol messages, interfaces between modules, and within modules themselves. Lastly, because it uses redundant edges and a standard metadata format, it supports incremental deployment in legacy environments. In the example of Figure 1, if server 2 were not instrumented, the client would still represent the edge from e_3 to e_8, effectively treating the server as a black box. Even when unable to instrument within that black box, reports can still reconstruct the latency of operations within it, as well as any erroneous return values or exceptions from its execution.

3 A Case for Causal Tracing

We now describe three systems in which causal tracing enabled us to determine the cause of faults and performance problems, find subtle bugs, and identify timing problems: an enterprise IEEE 802.1X deployment and two wide-area network services, CoralCDN and the OASIS anycast service. We present some of our findings here, and discuss the process of instrumenting the programs and protocols with X-Trace in the next section.

3.1 802.1X Authentication Services

The IEEE 802.1X network authentication service is a critical, distributed system for providing end-user access to network resources. Debugging it is a challenge, since failures at lower levels of the network stack (such as misconfigured firewall rules or high packet loss rates) manifest themselves as authentication failures at the 802.1X layer and thus prevent users from joining the network.

A successful authentication request typically involves the cooperation of at least four independent subsystems: (i) a *client* device that requests network access; (ii) a *authenticator* device that provides network access, such as a wireless access point, wired Ethernet switch, or VPN concentrator; (iii) a RADIUS server that decides if the client should be permitted on the secured network; and (iv) one or more *identity stores* that provide information about an organization's users or devices (*e.g.*, typically

LDAP directories, but also Kerberos, token servers, NIS, and databases). Often these components are managed by different administrative domains.

To better investigate 802.1X system failures, we partnered with a vendor of authentication network appliances. From their database of service requests, we identified five of the most common real-world error conditions that were non-trivial to detect and diagnose, and a small number of X-Trace instrumentation points were added to the authentication path that were sufficient to detect each of these faults. Given a reference graph of a "known good" authentication request, we can use abnormalities in the structure of collected graphs to indicate failures. This approach is a significant change from the common practice of piecing together hints of the root cause of failure from disconnected logs and protocol-specific diagnostic tools like ping and traceroute. We now briefly describe the features of the X-Trace graphs that we use to determine the root cause of authentication failures.

Misconfigured timeouts. We detect authentication timeouts by identifying the presence of both a timeout report (issued by the RADIUS server) at time T_1 as well as a report from the identity store at time $T_2 > T_1$.

Packet loss. Since the RADIUS protocol resides on UDP, packet loss between the authenticator and the RADIUS server results in the loss of an entire RADIUS request. We detect this by looking for paths in which there is an authenticator report but no corresponding RADIUS report. To detect loss on the reverse path, we look for X-Trace paths in which the RADIUS server sends a response to the authenticator at time T_1, only to have the authenticator time out at time $T_2 > T_1 + \delta$, where δ is greater than the one-way latency between the RADIUS server and the authenticator.

RADIUS Overload. A RADIUS server will reject RADIUS requests during overload, issuing a report to provide a deterministic signal that overload is the cause of the resulting 802.1X failure.

LDAP Overload. To detect LDAP overload, we apply a threshold test to the observed query latency, determining an error condition whenever the query latency exceeds 100ms. In our experience, any deployment with latency greater than this is exhibiting an error condition.

Misconfigured firewall rules. Network operators often put LDAP servers, which are considered "application" layer technology, in a different part of the network than the core network infrastructure. Thus, authentication requests often have to transit the firewall, leading to errors where a misconfigured firewall rule prevents authentication requests from completing. Since X-Trace optionally extends to the network layer, it is possible to directly detect loss due to firewalls, though we did not support those reports in our deployment.

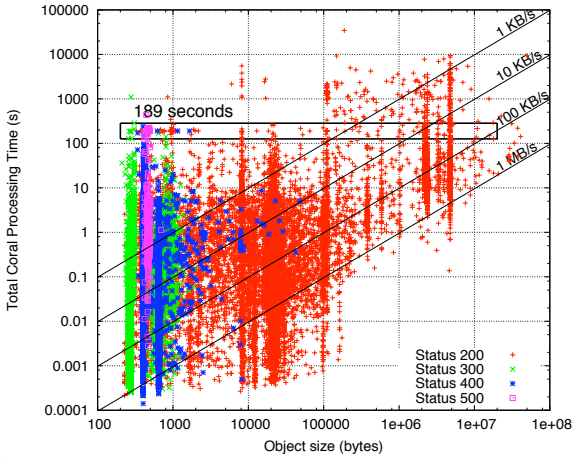


Figure 2: **CoralCDN processing time versus object size for 20,000 requests. Several different problems may have the same causes.**

3.2 CoralCDN and OASIS

CoralCDN is a popular open content distribution network, while OASIS is an anycast service used to select the best (*e.g.*, closest) replica of a client service. Each system is composed of multiple servers programmed using the event-based library `libasync`, and communicate among themselves in a distributed hash table-like (DHT) fashion using `libasync`'s RPC library or via HTTP. Externally, CoralCDN functions as a distributed HTTP caching proxy, while OASIS can be accessed either via DNS, RPC, or HTTP protocols. We instrumented live deployments on PlanetLab, and, unlike the 802.1X example, these systems' traces were complex, highly variable, and much larger (10s to 100s of vertices).

Coherent Sampling. We gathered data for this paper over two short periods for both services. Since we were severely limited in our collection infrastructure (we only used a couple of machines to gather the X-Trace reports), task-based sampling proved crucial. The 258 CoralCDN nodes we traced received a total of over 1.7M requests per day over 2.5 days, and we generated complete traces for a sampled 0.1% of these. For OASIS, we traced all 223,961 requests that the 19 servers received for one day.

Same symptoms, different causes. Figure 2 shows, for 20,000 CoralCDN tasks, the total response time given by CoralCDN versus the response's object size, also classified by HTTP status code. For each size, we see variations of 5 to 7 orders of magnitude. A deeper discussion on the many interesting phenomena present in this graph is beyond the scope of this paper, and we focus on a single group of requests. The rectangle in the figure shows several requests that took close to 189 seconds, for very different object sizes. Using X-Trace, we were able to examine complete traces for each point in the plot and to distinguish at least four different causes for the delays: a slow connection between the client and the proxy, a slow connection between the origin server and the proxy,

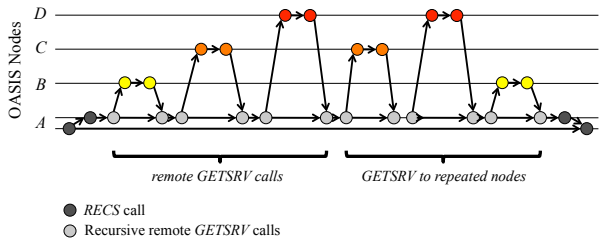


Figure 3: **X-Trace graph showing OASIS repeated nodes bug.**

a TCP SYN timeout when connecting to the origin server, and a large delay at a proxy due to it being de-scheduled by its PlanetLab kernel.

Bugs. We found a few bugs in both CoralCDN and OASIS, which had escaped other debugging techniques. We describe two such bugs that were easy to find using the task graphs, but hard to piece together otherwise.

In OASIS, an internal DNS record lookup is done through a RECS RPC call, which may invoke a series of GETSRV RPCs, the specifics of which are not important here. We found that in 3.3% of all resolutions, the calling node would issue remote GETSRVs to three nodes, which is normal, but then would issue repeated calls to the same three nodes in random order. These repeated calls were wasteful and guaranteed to fail, and they caused a 1.8× increase in response time. Figure 3 shows a simplified X-Trace task graph with the problem. While not impossible to identify using normal logging, this bug is easy to go unnoticed unless one is specifically looking for it.

The other bug caused CoralCDN to increase the load to an already overloaded origin server. In CoralCDN, many proxies can cache the same content. When a proxy A receives a request, it contacts one or more of its peers (say, B) that are caching the content. If proxy B's cached content is expired, B will first contact the origin server to revalidate it. If the origin server was taking too long to respond to B's request, then A could timeout waiting for B, and issue the same recursive query to another peer, C. Thus, as a whole, CoralCDN could issue one revalidation to the origin for each proxy storing a copy, certainly not within its goals of reducing origin load. Identifying this bug required correlating all revalidation requests at the proxies with the same original request: easy to do with causal tracing, but tedious with standard logging.

Tuning timeouts. As with 802.1X, we found X-Trace useful in tuning timeout parameters for specific RPCs, because with causal tracing we see *both* sides of an RPC. Specifically, if a RPC client times-out, we still see the server's execution (or lack thereof) and can judge whether the timeout was premature or too long.

Fixing time. Most distributed logging systems depend on synchronized clocks to correlate events. Not only does causal tracing not depend on synchronized clocks, it can correct unsynchronized clocks. For a pair of nodes, we can use any two messages in opposite directions (such as a request/response pair) and their local timestamps to detect clock inconsistencies, either online or in post-processing [16]. From our CoralCDN data, we found that 86% of nodes were synchronized to within 100ms, 95% to within 1s, and two nodes were close to a 1000s offset.

4 Instrumenting Systems with X-Trace

We have added X-Trace support to a number of systems. In addition to those described in the previous section, these include Apache, i3, Chord, Hadoop, DONA [14], and SCADS [2]. This involved modifying components written in C, C++, Java, Ruby, PHP and Javascript, and integrating with a variety of network protocols, including HTTP, LDAP, RADIUS, Thrift, DNS, IP, and Sun RPC. Extending these protocols to propagate end-to-end metadata (*i.e.*, X-Trace ids) not only provides increased visibility to that specific protocol, but also makes the higher-level applications—which use various combinations of these protocols—easier to reason about and debug. However, support *solely* at the network layer is not sufficient; applications must propagate appropriate metadata within and between processes, following the executing path. This section describes our experiences integrating X-Trace and some important challenges involved.

4.1 Propagation in Protocols

Communication protocols should have support for opaque metadata. This is already the case for many recent protocols. For example, HTTP has always supported extension headers, together with the requirement that implementations should propagate unknown headers unmodified when forwarding messages. Integrating X-Trace in the 802.1X authentication framework was straightforward, with each authentication request/response transaction represented by a since X-Trace task. For the RADIUS communication between authenticator and authentication server, we added a custom vendor-specific attribute (VSA) containing the X-Trace metadata. To instrument LDAP, we added a custom LDAP control. On the other hand, for some protocols we had to add X-Trace support in an ad-hoc manner that worked in practice but technically violated the specification. For example, when instrumenting `arpc`, `libasync`'s RPC library, we added the metadata after the payload of the message, which worked for both TCP and UDP transports in a backwards-compatible way.

4.2 Propagation within Programs

Propagating metadata inside programs is more of a challenge. At the most basic level, we need to take metadata from incoming messages/calls and carry it to causally-related outgoing messages/calls, while optionally logging events inside the application. In X-Trace, logging consists of creating a new event id and a report that binds the new

id with the immediately preceding event. It also changes the metadata going forward, as the new id has to be referenced by the next logged event. Thus, when logging an event, the code must have a reference to the X-Trace metadata which has the task and previous event ids.

The programming style of the application affects how easy it is to add metadata propagation. Perhaps the easiest style is that of request-oriented software with a simple internal execution path per object (*i.e.*, no parallelism), with well-defined hooks for event handlers that change the request object. A prime example is Apache, which maintains a context for each request and passes it around to dynamically linked modules that register callbacks. Adding X-Trace then is a matter of registering callbacks for the events of interest, such as receiving the request or sending the response, and adding X-Trace metadata to the (extensible) request context. Adding trace support to the LDAP component of the 802.1X framework similarly involved writing pre- and post-authorization "hooks".

For regular thread-based code, we wrote prototype libraries in C++, Java, and Ruby which have a per-thread global X-Trace context. Our libraries also hide all of the metadata manipulation and reporting from the programmer, and essentially expose a regular logging API. Internally, the logging call creates the new event id, writes the report, and updates the X-Trace context. The basic API consists of the following calls:

```
void xtr::setContext(xtr::Metadata m)
xtr::Metadata xtr::getContext()

void xtr::logEvent(string message)

xtr::Event xtr::prepareEvent(string message)
void xtr::Event::addEdge(xtr::Metadata)
void xtr::logEvent(xtr::Event)
```

A very different style of programming is event-based, such as in programs written with `libasync` or `libevent`. This continuation-passing style uses event handlers that schedule each other to govern program flow, *e.g.*, in order to avoid blocking I/O calls. The key to instrumenting such programs is to modify the event handler that registers and dispatches function calls, in order to save and restore the X-Trace context across "context switches" between disparate logical execution paths. Our instrumentation of `libasync`'s core changed four callback scheduling functions and the part of the event loop that dispatches callbacks, totaling less than 20 lines of code. The programmer can then always access information about the last event in the current logical execution path, using the same API as above.

4.3 Challenges and Experiences

While the instrumentation we have described so far covers most cases, there are some additional challenges.

Hidden Channels. One challenge is the use of *computational deferral structures* in systems. For example, it is not uncommon to add objects to queues for multiplexing and later processing. In this case, we want the metadata to be stored with the queue object, and restored when the object is processed. Sometimes control is passed between threads via shared memory structures, and metadata has to follow the control. Similar concern over hidden channels was made about CATOCS-based distributed systems [8]. Fortunately, these deferral structures are often part of libraries, making instrumentation easier.

We were able to detect instances of uninstrumented hidden channels by looking at graphs with unexpected structures, in an iterative process akin to the converging of instrumentation and expectations in Pip [18]. One example is the DNS resolver module in `libasync`, as used by CoralCDN. Initially, traces that included DNS resolution would always end at the event right before the call to DNS: the stateless DNS responses were not being matched with the X-Trace task that issued the request. The resolver, however, could demultiplex responses based on the DNS request id and port, stored in a hash table. The fix was to add the X-Trace metadata to table entries and to restore the proper context when returning control to the caller. In another example, CoralCDN would occasionally traceroute clients to detect network locality, but used a queue to limit the number of concurrent traceroutes. By extending the queued data structure to carry X-Trace metadata, we carried the appropriate context across this deferral.

Black-Box Tracing through Partial Annotations. It may not be possible to instrument some of the components involved in a task, either because source code is unavailable or the component runs outside of your control. For example, DNS servers and origin webservers played critical roles in CoralCDN, yet both were running unmodified in remote autonomous domains. However, it was still possible to capture their behavior in the X-Trace graphs by treating them as black-box components that X-Trace traced around, as opposed to through. X-Trace reports can still record timing information, as well as the existence, type, and return value of responses. Of course, the root causes of latency or errors of the subsystem are not visible if it is uninstrumented.

If the proxy in Figure 4 has no support for X-Trace, we can always trace around it, recording events at the client only. If it can propagate metadata, even without understanding it (like HTTP proxies are required to do, for example), we can trace through it, correlating events at the client and the server. Lastly, if it fully supports X-Trace we get events from all three components. X-Trace requires no coordination, and different parties can instrument the three components, provided they agree on a way to exchange their X-Trace reports.

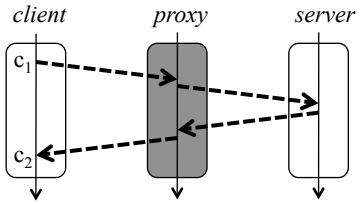


Figure 4: **Tracing *through* a black-box proxy (as depicted) depends on whether the proxy propagates metadata. Even if it does not, we can always trace *around* it at the client, establishing the causal relationship between c_1 and c_2.**

Semantic vs. Incidental Concurrency. Another challenge is to capture concurrency correctly. In the case of several threads logging concurrently, X-Trace distinguishes the incidental ordering due to how they are scheduled from the true dependencies. However, there are challenges when concurrent events in logical threads interact.

Figure 5 shows a hypothetical program that starts three parallel function calls in line 5 and implements a barrier in line 12. If we use the standard X-Trace instrumentation by just recording the relevant events, we get the graph at the bottom of the figure. While the forks are technically correct (a fork is always a bifurcation in the program flow), the join is misrepresented: the *end* event is connected to the last *done* event to finish, even though it depends on all three. Currently, X-Trace requires manual annotation in the *end* event to represent this, as shown in Figure 6. For the fork, line 6 sets the preceding event to the *start* event. For the join, we first create an event in line 14, add the edges in line 16, and finally log the event in the last completion, in line 18. The programmer has to keep around the ids of all events that are waited for in the barrier, and record all edges in the last one. It is an open question how to do this automatically.

Task-based Severity and Sampling. We described how X-Trace allows for easy coherent sampling. The first logical event in a task can decide whether the trace the entire task (*i.e.*, the sampling decision); later in the task flow, we first test whether a valid context exists before generating an X-Trace log event. Another mechanism we found valuable is *severity-based reporting*. Like common logging frameworks, each log statement in X-Trace has a severity level, and the runtime has a global logging severity threshold. A statement is logged if its severity is above the global threshold. X-Trace metadata can include a per-task severity threshold that overrides the global threshold, thereby enabling per-task logging detail.

Expectation-Driven Analysis. Our analysis of the 802.1X service used a notion of a "ground truth" graph structure, and other LAN or intra-datacenter services have low-latency expectations that, when exceeded, signify some type of fault. In both cases, it is easy to automatically flag particular traces as exhibiting anomalous

```
 1:  const int N = 3;
 2:  xtr::logEvent("start");
 3:  for (i = 0; i < N; i++) {
 4:      xtr::logEvent("do(%d)", i);
 5:      doSomething(i);
 6:  }
 7:  int remaining = N;
 8:  void somethingDone(int i) {
 9:      xtr::logEvent("done(%d)",i);
10:      if ( --remaining == 0 ) {
11:         xtr::logEvent("end");
12:         done();
13:      }
14: }
```

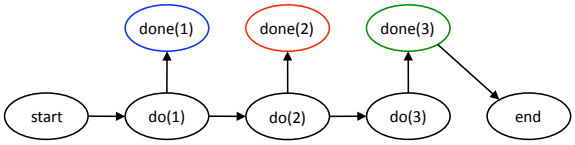


Figure 5: **Standard X-Trace instrumentation (in bold) for a fork/join pair. In the resulting X-Trace graph, the fork looks unintuitive, and the join is not captured correctly.**

```
 1:  const int N = 3;
 2:  xtr::logEvent("start");
 3:  xtrMetadata start = xtr::getContext();
 4:  for (i = 0; i < N; i++) {
 5:    xtr::logEvent("do(%d)", i);
 6:    xtr::setContext(start);
 7:    doSomething(i);
 8:  }
 9:  int remaining = N;
10:  XtrEvent xte = null;
11:  void somethingDone(int i) {
12:    xtr::logEvent("done(%d)",i);
13:    if (xte == null)
14:     xte = xtr::prepareEvent("end");
15:    else
16:     xte.addEdge(xtr::getContext());
17:    if ( --remaining == 0 ) {
18:      xtr::logEvent(xte);
19:      done();
20:    }
21:  }
```

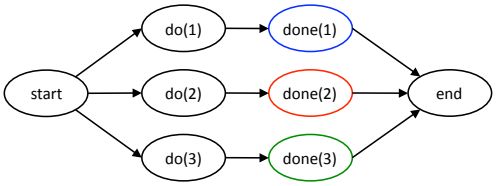


Figure 6: **Modified instrumentation that captures the fork and the join correctly. For the fork, we have to reset the context to *start* after each fork, and for the join, we have to manually add the extra edges to the join event.**

behavior, and then perform subsequent manual analysis of these traces. Tracing the wide-area CoralCDN and OASIS, however, did not similarly enjoy such easily classifiable behavior, *e.g.*, traces with repeated queries (Figure 3) often experienced lower execution times than correct behavior involving distant peers. Similarly, the behavior of system components would vary greatly from request to request, depending on the state of both local and remote nodes (*e.g.*, in CoralCDN, whether the requested URL was cached locally, required a DHT lookup or revalidation, was fetched from a nearby or distant peer or from the origin, etc.) Yet when faced with 100,000s or millions of traces, purely manual inspection is not feasible, and we must rely on automatic, unsupervised detection schemes, which we leave as future work.

5 Related Work

A counterpoint to causal logging are device-centric management and monitoring solutions such as SNMP or Nagios. Correlating log entries across these types of single-node logs is a challenge, and approaches such as Splunk [19] allow searching across multiple log streams. In [20], the authors use machine learning with source-code analysis for anomaly detection in console logs.

We are not the first to argue for metadata propagation as a valuable primitive, and others include Causeway [6] and SDI [17]. Many other systems [5, 7, 17, 18] provide different levels of causal tracing, anomaly detection, or profiling using propagated metadata to correlate events. Whodunit [5] also integrates metadata propagation to `libevent`-based systems and across some shared-memory structures, and it uses the information to build a profiler that spans components of a distributed system. Magpie [4] does causal tracking without using any annotation, but requires extensive domain knowledge and logging. Microsoft's Event Tracing for Windows provides cross-component tracing that can propagate activity ids across Windows-based systems. Pip's iterative expectation-refining model is similar to what we did in practice when finding errors both in the code and in our own instrumentation.

Other systems avoid modifying applications or adding metadata, and have limited visibility into the system. BorderPatrol [15] uses knowledge about protocols and assumptions about application behavior to correlate inputs and outputs to system components treated as black boxes. Project5 [1], Sherlock [3], and NetMedic [13] take an inference-based approach and try to find performance anomalies or the root cause of problems.

6 Conclusions

In this work, we describe our experiences adding support for tracing causality in real-world applications. Despite the diversity of applications, protocols, and programming runtimes employed by these case studies, we find that a simple, uniform graph data structure is sufficient to detect a variety of error conditions. These include network-level faults like firewall misconfigurations and packet-loss events, to network service failures in the case of DNS, to detecting timing and clock synchronization issues at the application layer. X-Trace has served as a single, coherent view of the state and history of a composed application, making the management of that application across increasingly complex network architectures tractable. Our experiences have shown that instrumenting non-trivial systems is not only possible, but often straightforward.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.

[2] M. Armbrust, A. Fox, D. Patterson, N. Lanham, H. Oh, B. Trushkowsky, and J. Trutna. SCADS: Scale-independent storage for social computing applications. In *CIDR*, 2009.

[3] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. *SIGCOMM CCR*, 37(4):13–24, 2007.

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modeling. In *OSDI*, 2004.

[5] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *Eurosys*, 2007.

[6] A. Chanda, K. Elmeleegy, A. L. Cox, and W. Zwaenepoel. Causeway: System support for controlling and analyzing the execution of multi-tier applications. In *Middleware*, 2005.

[7] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem determination in large, dynamic, Internet services. In *DSN*, 2002.

[8] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. *SIGOPS OSR*, 27(5):44–57, 1993.

[9] R. Fonseca. *Improving Visibility of Distributed Systems through Execution Tracing*. PhD thesis, EECS, U.C. Berkeley, Dec. 2008.

[10] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *NSDI*, 2007.

[11] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *NSDI*, 2004.

[12] M. J. Freedman, K. Lakshminarayanan, and D. Mazières. OASIS: Anycast for any service. In *NSDI*, 2006.

[13] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. *SIGCOMM CCR*, 39(4):243–254, 2009.

[14] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *SIGCOMM*, 2007.

[15] E. Koskinen and J. Jannotti. BorderPatrol: Isolating events for black-box tracing. In *Eurosys*, 2008.

[16] V. Paxson. On calibrating measurements of packet transit times. *SIGMETRICS Perform. Eval. Rev.*, 26(1):11–21, 1998.

[17] J. Reumann and K. G. Shin. Stateful distributed interposition. *ACM TOCS*, 22(1):1–48, 2004.

[18] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, 2006.

[19] Splunk. `http://www.splunk.com`.

[20] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Large-scale system problem detection by mining console logs. In *SOSP*, 2009.