DCPT: Distributed Critical Path Tracing for Latency Profiling

Leobardo Lopez
Stanford University

Will Song
Stanford University

Veerendranath Mannepalli
Intel Corporation

Abstract

Latency profiling in distributed systems is critical for site reliability, service robustness, and fairness. However, traditional latency profiling techniques have high overhead and collect data aggregates that usually monitor a specific service at a single node. Critical path tracing can be applied to diagnose performance issues faster with minimal disruptions in large scale distributed services foregoing complex and invasive infrastructure.

We implement Distributed Critical Path Tracing (DCPT) as a low latency and request-conscious form of latency tracing. Our implementation is based in XDR [2] and exposes a minimally-invasive API for performance tracing. We test our implementation with simulated workloads and analyze time and space overheads in extreme circumstances.

1 Introduction

Modern computing services are served on large and complex distributed system topologies with hundreds of nodes and many parallel links between them. These large networks make latency profiling obscure due to the large volume of RPCs servicing requests. Service providers find themselves employing tens of engineers for monitoring the performance of these services. These performance engineering teams typically use a combination of telemetry, instrumentation, and distributed tracing to form some heuristic of overall system health (latency and load). Common profiling techniques are either service-node local (instrumenting machines or logging RPC boundaries), or request local (distributed tracing). However, we find both of these techniques do not present actionable outputs for a performance engineer.

Throughout this paper, we consider techniques as either request facing (information clients issuing calls can directly see) and node facing (information local to servers that administrators can see). Section 2 introduces some related works and preliminary work on Critical Path Tracing. Sections 3 and 4 introduce the implementation of DCPT on XDR. And Section 5 presents overhead evaluation.

1.1 Background

1.1.1 Instrumentation

Traditional performance instrumentation systems are usually node facing and involve some form of server level tracing through either RPC telemetry or hardware profilers [4]. Available distributed tracing infrastructure generated traces are typically collected by a background performance daemon that collects performance counters and coalesces RPC traces to export them to a health check in a periodic schedule. These techniques produce actionable performance information hours, but at best it will be after an event has occurred. Since the traces need to be stored on servers and be retrieved at a later time, there is a certain space constraint to the size of the traces in instrumentation, typically forcing systems to sample traces and aggregate them. Here, we easily lose information about the 99th percentile of request latencies.

1.1.2 Distributed Tracing

Modern distributed tracing methods such as Dapper [7] collect all the RPCs of a particular request to paint a full picture of all the sub-components involved in the request. They can be request facing and return specific information tagged along packets to system administrators or performance teams if they send special requests. However, these traces record all outbound RPC calls and all inbound RPC replies. Additionally, most of these sub-components are not the central bottleneck for a response. Modern computing centers have an extremely high degree of parallelism and request response is solely determined by the slowest component. Hence, many engineers have to look at these logs of traces for a while before finding even the first three nodes of the critical path.

2 Related Work

We are implementing the Distributed Critical Path scheme inspired by [1]. Most large distributed services use some form of health check or telemetry to monitor request load balancing.

For example, Amazon has their own Cloud Watch Service [6] and Google has Dapper [7]. Although these works are related, performance monitoring infrastructure is typically in-house with particular performance feature depending on the service type.

2.1 Dapper

The closest neighbor to DCPT is the Google distributed tracing Dapper [7]. It is a Low Overhead distributed out-of-band trace collection infrastructure used to trace individual requests through a distributed system to highlight latency bottlenecks. The trace is stored in local log files, collected later and transported to a central repository.

Dapper is a request facing infrastructure. The request issuer can find the characteristics of the request they sent by querying for the trace upon completion. Most users can tag a request, have it monitored, then find the trace on a central big table. It also has low overhead and a transparent API - most engineers don't need to know it exists to properly interact with its services. The key differences between Dapper and DCPT are the following:

First, DCPT uses in-band trace collection where the trace tree is sent back in the RPC response body. Second, the volume of information captured in the traces. In a sense, Dapper contains all the information DCPT would contain. However, the engineer would need to look through all the RPC links to find the critical path, which slows down performance pin-pointing. Moreover, Dapper is implemented with a background Daemon on Dapper-enabled machines to feed information to a centralized set of Dapper collectors and a big table to store the data. This information can take time to surface and the engineer will have to wait. We hope with the reduction in size of traces (by about a log factor), DCPT can omit the background daemon and centralized trace collector.

2.2 X-Trace

For node local profiling techniques, some techniques in X-Trace [3] are popular choices. X-Trace is a network tracing framework that focuses on recording the path a request takes through the sub-components of the network stack across multiple applications inside a specific server that abides to administrative boundaries. It implements an in-band trace collection strategy by constructing the trace by flowing along the data-path through the whole network stack. DCPT differs by constructing paths across RPC protocol boundary and tracing across the server tier nodes. X-Trace stores the trace path out-of-band within the node itself under the control of the administrator, whereas DCPT attaches the trace path to the response RPC body and propagates it across the nodes until the specific request is completed.

Although X-Trace is used in many scenarios, it only provides the path data. This means other monitoring tools will

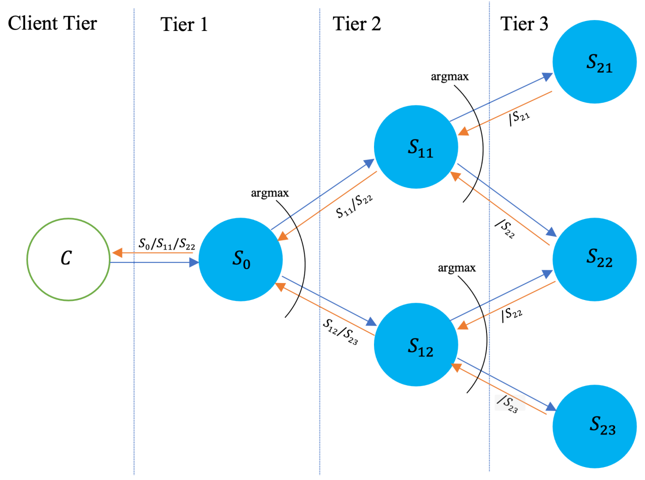


Figure 1: An example implementation of CPT in a three layer network topology divided into tiers serving a request a single client. Each node takes the next tier's returned paths and latency, then propagates up the argmax plus its own time. The final returned path for the client here is $S_0/S_{11}/S_{22}$.

be needed for latency info to find the bottleneck for a specific request within the trace. Since it stores traces within individual node, more engineering effort is required to pull out the traces of each request and start putting the pieces together across nodes to construct critical path.

3 Methods

We define a network topology as $T = (G, E)$, where G represents the set of nodes in our topology and E is the set of communication channels. We track the Critical Path per-node by propagating back only the argmax of the downstream paths for each level in our tier system. For example, in Figure 1, we show a top-down tiered server with $|G| = 6$ and $|E| = 6$. The critical path here has size $\lceil \log_2(6) \rceil = 3$ which is the size of the deepest path.

Let $S_{t,s}$ be a server s at a tier t. $S_{t,s}$ will return a path $p_{t,s}$ with:

$$p_{t,s} = S_{t,s} \oplus \underset{p \in \{p_{t-1,s'} | s' \in \text{next tier}\}}{\arg\max} time(p)$$

We use $\oplus$ here to denote path concatenation. With this, we construct a critical path for some node $S_{t,s}$ using the critical path of the children servers (in the next tier). Finally, $S_{t,s}$ will propagate back a tracing tuple $(p_{t,s}, time(p_{t,s}))$ where $time(p_{t,s})$ is taken independently of any children times to account for computation time spent in $S_{t,s}$.

The beauty of the node-wise argmax is that the overall path we receive at the client response includes every critical latency edge for the request. Hence, if we improve any channel $e \in$

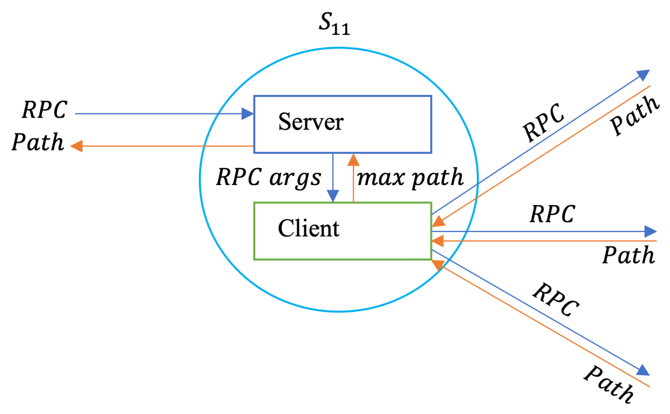


Figure 2: An example implementation of tracing around a critical path within a single node.

$p_{0,0}$, we will see at least some improvement in the maximum latency.

4 Implementation

We implemented DCPT on top of xdrpp, a C++ RPC library created by David Mazières [5]. xdrpp is built with XDR [2], which is used to encode data and communicate between multiple machines . We add new fields to the RPC message, modify the xdrpp stack, and minimally require a few variables at the top level server client implementation. We also implemented a Distributed Tracing Profiler (DPerf) for comparison using the same stack as DCPT.

4.1 Modification to RPC Packets

4.1.1 RPC Call Requests

We add a new field to the RPC Call request that tracks the mode the client requests, namely a TraceMode. This is reflected in the RPC signature for the tiering server that is exposed to the top layer as an API.

```
RPCinvoke(args, cb_func, trace_mode)
```

where `trace_mode` is an optional flag that enables DCPT tracing for the particular request. We trace this flag throughout our servers using a field in the RPC call header:

```
hdr.body.cbody().trace_mode = trace_mode;
```

Incoming packets with `trace_mode` specified will receive a reply with the associated trace. To initially set the `trace_mode`, the client can issue command line args: "-t" and "–tracing" for DCPT, whereas "-d" and "–distributed" to specify DPerf. By default, tracing is turned off. There are three possible TraceModes: {OFF, TRACING, DISTRIBUTED_TRACING}. Example usage is:

```
./async_client [-t | -d] key value
./async_client [--tracing | --
distributed] k v
```

Unless an RPC is specifically targeted, most functions will not be traced.

4.1.2 RPC Reply Responses

We added two new fields to successful RPC reply responses: a variable-sized path field and a time variable indicating request completion. These fields track the current state of our trace. In particular, when a request is completed at some node B, we need to forward the current tracing state up to parent node A. Note that node B must account for tracing information from its children nodes to have an accurate DCPT at its node-level.

We store the end time using a CycleTimer, a processor cycle-counter borrowed from CS149 that tracks time. We compute the difference (in seconds) between the time an RPC Call request is dispatched and when the reply callback is called. This tracks the total time it took for the request to complete, which can be relayed to parent nodes.

4.2 Modification to xdrpp stack

xdrpp registers a server to an asynchronous RPC TCP listener when a server is spawned up in order to process RPC requests as the server receives them. When a request is dispatched, we modify the session associated with the dispatched request to include the node name and the trace mode. We modify the server type such that it includes a node name on the instance itself. The trace mode exists as a global variable per node instance, which is initialized on dispatch by reading the RPC Call request.

Recall that timing is captured from the moment an RPC call request is made and when the reply callback is called. We accomplish this by overloading `reply_cb::operator()`, with our tracing functionality. This functionality includes calculating the time as detailed above, and constructing the path for this node. This operator overload was used by xdrpp to send replies back to the client, which is the exact endpoint our tracing requires.

When a reply is sent, we extract our tracing data through a global tracing variable, which is a pair of path and time. This node-instance variable gets modified each time reply callbacks are accessed via operator() so that each node has enough information for its current tracing state. At this step, we have enough information to properly forward a reply to the parent node with the original XDR reply fields and our own fields, namely path and time.

4.2.1 Asynchronous Requests

Our tracing implementation supports asynchronous RPC requests. To calculate the critical path of a particular transaction, we only record the critical path up to the point of issuing a reply. This is because the end client only cares about the latency of a response. Hence, we would only need to trace the critical path of the set of callbacks directly considered for a response.

We reflect this in our tiering system by using a blocking counter at each tier server to wait for all lower tier server responses (we consider all replies). All requests are still asynchronous and can be executed in parallel. However, we need to wait for a response from all of them. Then, we take the argmax of children servers' times and reply with our critical path.

4.2.2 Server Client Synchronization

Each node instance also has a mutex associated with its tracing state. When a node calls `reply_cb::operator()`, we create a critical section around our tracing logic. Global variables are used to track the tracing state, such as max path constructed from the child nodes and times associated with the paths. Some internal structures that store tracing information involve maps from xids to time and xids to path. This mutex protects the maps from being overwritten as well as modifying the per-node tracing state.

Generally, bottlenecks are introduced with synchronization (especially for performance tracing). However, DCPT can make use of service internal synchronization. In our case of a distributed key value store, we need to use a mutex to coalesce responses anyway, hence, the overhead should be reasonable.

4.3 Server Types

Our tier server model contains two kinds of servers: Tiering servers and Data servers.

4.3.1 Tiering Server

The responsibility of the tiering server is to connect other servers and create a topology through a tiering system. Our tiering system is set up with the following command:

```
./tiering_server offset [offsets ...]
```

Note that `offest` and `offsets` are unique offsets from a base port, namely 30428, for API simplicity reasons. The tiering server first opens a TCP listening socket by binding to a unique port (`30428 + offset`), allowing it to receive dispatch messages from the parent servers. It also opens TCP connect sockets for the children servers (with ports `30428 + offsets`) in order to forward requests onto them during the bring-up phase. Both the listening socket and the connect sockets remain open until the tiering server is terminated.

Each tiering server has a `unique_id` string that is based on the unique port it listen on. Our tracing implementation uses this id during the reply phase when constructing the distributed or critical path.

4.3.2 Data Server

The responsibility of the data server is to complete the RPC request and reply to the client. This server is set up with the following command, where `offset` specifies a unique port offset from our base port:

```
./async_server offset
```

Recall that we use a tiering server to create our network topology. In particular, this means that the client of a data server is a tiering server. Once the reply is constructed from the data server, it gets propagated up along the parent tier servers until it reaches the original, non-tiering-server client. Our tracing implementation calculates the critical path(s) at each node along the reply's propagation upwards. Each tier looks at the path and time fields specified in the reply, and modifies it according to our tracing specifications to construct a new trace, which is continually propagated up the network topology. The client base API looks like this:

```
void async_base(args, cb_func, trace_mode)
```

Where trace status is traced by the same field in the RPC call packet:

```
hdr.body.cbody().trace_mode = trace_mode;
```

4.4 Tier Server Model

For simplicity, we have three tiers of servers in our implementation (see figure 1). Tier 1 contains a single tiering server that receives clients' requests and forwards them to tier 2 tiering servers. Each Tier 2 server receives the data request from tier 1 server and forwards the data request to tier 3 data servers. These data servers process the RPC request and construct a reply. Note that this server also contributes to the tracing information we collect.

When the reply is completed, it is sent to the client, which is a tier 2 tiering server. This tiering server looks inside the RPC reply and extracts the tracing information in order to compute the critical path *for its current node*. Note that this node's critical path will be dependent on all children servers it is connected to. For example, S_{11} will observe the tracing information of S_{21}, S_{22} to determine which tracing information to append to, such as the critical path. This process repeats for the transition to Tier 1 and the client tier as well. When the client receives the reply, it will also receive the tracing information.

5 Evaluations

We primarily compare our DCPT implementation against a control system with no profiling and DPerf (also implemented in xdrpp using a similar method). We first evaluated the quality of a critical path in the same 3-tier key value store served

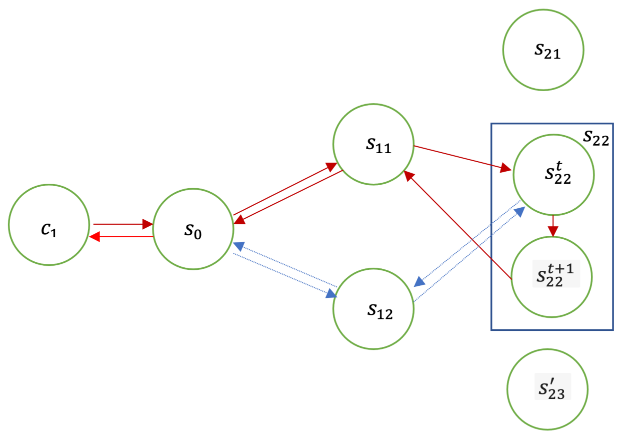


Figure 3: A hidden critical path (red line) on a single threaded node. Server s_{22} is represented with the blue box and its computation at time t is denoted as s_{22}^{t} to represent the calculation of another node's request (dotted blue line). The critical path response is handled at time $t+1$ denoted by s_{22}^{t+1}.

on the original implementation of xdrpp. We then moved to scaled evaluation on a chained server architecture and a fanned tiering architecture of size 10, 100 and 1000 in order to measure metrics such as latency (Figure 4).

5.1 Testing Infrastructure

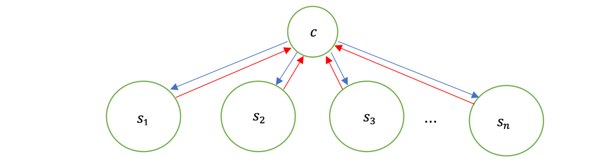
We ran all of our tests on a single AWS EC2 g4dn instance with 4 vcpus spawning the chained server and fanned servers in Figure 4. Each node is serviced on a thread, so we effectively made 1000 threads to simulate 1000 nodes on AWS. Our synthetic work load is a set of 10 $kv_{p}uts$ and simulated wait times in each of the nodes. For statistical accuracy, all of our evaluations are done across 10 `kv_put` requests and we report the average.

5.2 3-Tier-Implementation

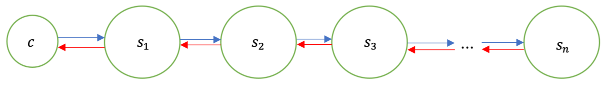
To demonstrate an interesting path and show our implementation extracts a critical path from a meaningful topology, we first implemented the 3-Tier server as show in Figure 1. We simulated a key value store workload with end nodes s_{21}, s_{23} taking a 1 second delay for a computation (sleep for 1 second) and s_{22} taking a longer 3 second sleep. All the servers are single threaded.

We found an interesting path on the 3-tier where two parallel computations were forced to execute sequentially (since our servers were single threaded) and it created the critical path (Figure 3). Specifically, our critical path returned the following:

```
critical path: "Server0_[6.016368s]
/Server11_[6.015887s]/Server22_[6.015142s]/"
```



(a) Fanned server topology



(b) Chained server topology.

Figure 4: Two evaluation server types for overhead of DCPT and DPerf. Red lines show tracing directions (for both DPerf and DCPT).

In this scenario, the Distributed tracing would return every RPC issued and the end engineer can repiece together the 3-nodes. However, we see this is slow and unscalable if we have more than 3 nodes. Worse, from the perspective of each node, computation is fine. s_{22} does not care that the request took 6s because it was always serving some request (at time t and $t+1$, s_{22} was never idle and the CPU functioned as normal). Hence, this makes it extremely hard for server facing traces to show the response to this request was slow due to congestion at s_{22}. If the request is a common, the performance team could infer workload imbalance if server s_{22} is always at full CPU utilization. But requests like the one in red can be rare and will be hard to discover.

5.3 Performance

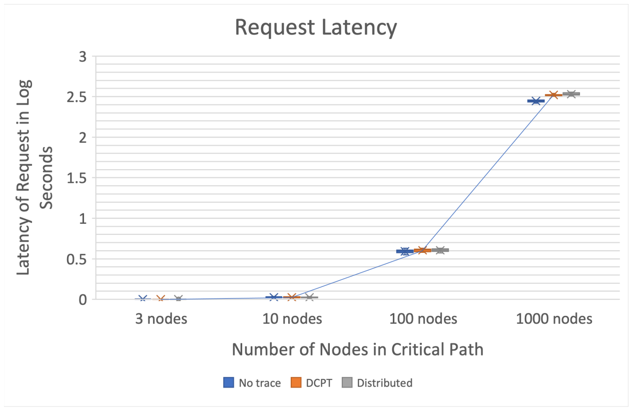


Figure 5: Latency overhead of DCPT vs Distributed Profiling and an untraced control on the chained server architecture.

	3 nodes	10 nodes	100 nodes	1000 nodes
DCPT Chain	0.99x	0.99x	1.03x	1.18x
DPerf Chain	1x	0.99x	1.03x	1.21x
DCPT Fan	1.6x	1.09x	1.15x	1.21x
DPerf Fan	1x	0.99x	1.26x	1.5x

Table 1: Average timing overheads by multiplier for DCPT and Distributed Profiling. We don't really see very significant overheads on chained architecture until about 1000 nodes which has around 20% overhead. On the fanned architecture, we see large overhead due to computation from 3 nodes and on.

5.3.1 Latency Overhead

Since we are implementing a tracing method, overhead is extremely important. To simulate a workload where tracing information could lead to congestion, we implemented a chained server (Figure 4) to incur overhead. The log latencies are shown in figure 5 and the average overhead latency are show in Table 1 as a multiplier. Our implementation starts suffering significant overhead at the 1000-node chained structure. At 1000 nodes, we see the trace starts becoming like 20k bytes long (our per-node trace is roughly 20 bytes long). This is quite long and we see a near 18% latency overhead for DCPT. Since we implemented DPerf to track mainly internal RPC's, it returns a similarly sized trace (since there is only one path in the chain). But it has a higher overhead for accounting for each local node.

Performance on the Fanned server topology highlights both the cost of DCPT on small topologies and DPerf on larger topologies. The network time for the Fan architecture was on the order of hundreds of microseconds since the architecture is shallow (two hops). Hence, the actual computation of a critical path within each node could become a bottleneck. However, we see the DPerf performs worse at scale after the network topology contains more than 100 nodes. This is intuitive since the size of a DPerf trace for $n = 1000$ includes all the paths (roughly 20 k bytes) and DCPT only returns a path with 40 bytes (critical path here has two servers since the whole topology has two hops.). In general, we can expect DCPT to return a trace on the order $O(\log(|E|))$ since the critical path would hopefully not include every possible edge or cycles.

5.3.2 Space Overhead

Both our DCPT and DPerf implementation save no extra overhead within nodes (except for entries in a global map on memory, which is recaimed on server shut down. The main space constraint is within the XDR packets containing the traces. In the chain implementation, DCPT generates a trace the size of $O(n)$, a constant addition per node in the n sized critical path. DPerf generates a trace size of size $O(n)$, but with a larger constant since each node also attaches its own local trace. This is expected since the workload is on a single chain.

However, in the fan-out architecture, DCPT only generates a trace of size $O(1)$ since the critical path only has 2 servers, but DPerf generates a trace of size $O(n)$, one for each node. In this scenario, DCPT is better in space.

6 Conclusions and Future Work

We implemented DCPT and a DPerf system on XDR to trace per-request critical paths from a relatively new paper ([1] was published in March 2022). We saw a clear advantage against traditional DPerf for request facing traces, both in terms of trace size and latency scaling. Moreover, DCPT is workload sensitive since it can adaptively adjust sampling frequency via optional parameters in RPC call signature. DCPT acts as a transparent performance monitoring layer that the application programmer can ignore until they need actionable performance information. Then DCPT can provide a quick and actionable critical path trace. This should expedite Performance Engineering work and help scale large products.

One potential future direction is making DCPT compatible with a background daemon for data collection. If a service has a small number of nodes, but each RPC contains a critical path greater than length 1k (this can happen if an RPC loops over all nodes in a cluster several times), then the critical path traces themselves become a bottleneck and would need to be offloaded in the background. Hence, making DCPT compatible with logging and tracing infrastructure is a good future direction.

References

[1] Brian Eaton, Jeff Stewart, Jon Tedesco, and N. Cihan Tas. *Distributed Latency Profiling through Critical Path Tracing*. ACM Queue, 2022. https://queue.acm.org/detail.cfm?id=3526967.

[2] Ed. Eisler, M. XDR: External Data Representation Standard. RFC 4506, RFC Editor, 5 2006.

[3] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. X-Trace: A pervasive network tracing framework. Technical report, Cambridge, MA, April 2007.

[4] Intel. Intel® vtune™ profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[5] David Mazières. XDRPP. https://github.com/xdrpp/xdrpp, 2022.

[6] Amazon Web Services. Amazon cloudwatch. https://aws.amazon.com/cloudwatch/.

[7] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.