Distributed Tracing For IPFS

Sushant Kumar Gupta, Marshall David Miller, Rachel Han, Haorui Guo
Stanford University, gsushant, millermd, rachelyh, haorui@stanford.edu

***Abstract* - We aim to gain valuable insights into how requests propagate within a P2P network as well as identify potential performance bottlenecks which could lead to opportunities for improvement.**

We propose Nabu-Tracer, a system that traces important events made within an IPFS network. We have implemented a custom tracing framework on top of an existing IPFS implementation. Our framework creates and manages traces for the various events that happen when retrieving content from the network.

Concretely, we augmented and worked with a minimal IPFS implementation written in Java called Nabu.

Introduction

The *InterPlanetary File System* (IPFS) [1] is a decentralized protocol that facilitates the publication and retrieval of content. Network participants within the IPFS ecosystem are called peers. Peers leverage a distributed hash table (DHT) to orchestrate the placement of content records.

Primarily, IPFS supports two APIs:

- *PUT*: This function writes the provided content to the local storage and subsequently publishes the location of that content, i.e. the peer record, on the DHT. The function returns a unique content identifier (CID) to the calling client.
- *GET*: This function retrieves the content associated with a given CID.

The GET request leverages two separate protocols. First, it is necessary to traverse the DHT in order to locate a peer that possesses or *knows* another peer who possesses the requested content. This traversal utilizes the *Kademlia* [2] lookup protocol. The second protocol used by the GET request in IPFS is the *BitSwap* [3] protocol. *BitSwap* is responsible for the actual content fetch operation. Both *Kademlia* and *BitSwap,* require the execution of multiple network operations in order to coordinate with other peers in the IPFS network.

IPFS is anticipated to evolve into a foundational element of web3 technology. Primarily, IPFS is the storage layer of web3 content. An issue with IPFS (and the inherent nature of decentralized applications) is the fact that the network is susceptible to the presence of sluggish peers. Slow participants can hinder content retrieval. In the context of IPFS, it would be of significant interest to identify slow participants as well as identify other network bottlenecks. To address this need, we present a novel tool that, when paired with a user interface (UI), allows users and developers to visualize network traces generated during the content retrieval process.

Related Work

Our project was initially inspired by workflow-centric approaches that have been pioneered by systems such as Google Dapper [4], Cloudera HTrace [5], and Meta Canopy [6]. These systems leverage end-to-end tracing to meticulously capture the workflow of causally related activities across the diverse components that comprise a distributed system.

X-Trace [7], another workflow-based system, has achieved success in network monitoring and application tracing. However, X-Trace necessitates instrumentation at numerous points of interest within the underlying system. In contrast, our approach requires instrumentation solely within the Threading and RPC libraries.

OpenTelemetry [8] is another open-source, state-of-the-art tracing solution, which provides automatic tracing instrumentation as well as comprehensive manual instrumentation capabilities. However, its automatic instrumentation can only monitor the edge of a system, such as inbound and outbound HTTP requests, but doesn't capture application's semantics.

For trace processing, we leverage the inherent semantics of the events to infer causal relationships. A closely related system is Magpie [9], a tracing tool adept at processing events generated by both operating systems and application instrumentation. Magpie, too, relies on the

Table 1: Events captured during tracing.	
Event Type	**Description**
GET_PROVIDERS_CLIENT	When a Kalemdia lookup request begins/ends on the client side.
GET_PROVIDERS_SERVER	When a Kademlia lookup request begins/ends on the server side.
BITSWAP_CLIENT	When a BitSwap request begins/ends on the client side.
BITSWAP_SERVER	When a BitSwap request begins/ends on the server side.
READ_FROM_FILE_STORE	When the content retrieval begins/ends from the store.

events themselves to establish causal relationships.

DESIGN

The process of tracing a GET request within the IPFS network can be segmented into three distinct stages:

- ***Sampling***: Due to scalability constraints, not every request received by the IPFS network can be subjected to tracing. A designated sampling policy decides which requests are chosen for tracing. If a request is selected for tracing, a unique trace identifier is generated and subsequently propagated throughout the network.

A strategically advantageous location for implementing the sampling function resides within the gateway servers. Gateway servers, while distributed across the Internet, can be configured to access a central policy repository. This repository would then store the state of the sampling policy.

- ***Event Capture***: For each request that carries a trace identifier, a specific set of events are captured, as outlined in Table 1. These events, paired with their corresponding start and end timestamps, constitute *spans*, which serve as the fundamental building blocks of a trace.

Each span is enriched with additional metadata to establish causal relationships between events. Particularly, a span comprises the following fields:

- *Node Id:* Identifier for the node that exported the span.
- *Thread Id:* Thread identifier for the thread associated with the event execution. This field is primarily used for debugging purposes.
- *Peer Node Id:* This field identifies the peer node involved in the event. It plays a crucial role in establishing relationships between spans. For instance, the GET_PROVIDERS_CLIENT span will contain the Id of the peer server that received the request. Similarly, the corresponding GET_PROVIDERS_SERVER span will include the Id of the peer client that sent the request.
- *Debug String:* This field allows for the inclusion of supplementary information pertaining to the event.

- ***Export & Aggregation***: Following their capture, the spans are exported to a backend system. This backend system consolidates spans associated with a particular trace identifier. Prior to persisting the trace data for retrieval purposes, causal relationships between the spans are meticulously resolved. Figure 1 illustrates the algorithm employed to establish these causal relationships between spans. In essence, the algorithm focuses on determining parent-child hierarchies among the spans within a trace. This process leverages the inherent semantics embedded within the spans themselves.

IMPLEMENTATION

I. ***Nabu***

Nabu [10], a minimalistic implementation of the InterPlanetary File System (IPFS), facilitates data block storage and retrieval. We have augmented Nabu with a tracing layer. This augmentation necessitated several key modifications:

- The GET API was adapted to accommodate trace requests. A trace Id is generated by the peer which re-

ceives the GET request from the HTTP gateway.
- The trace context was propagated across threads within a peer and across peers over the network. For the latter, the RPC stub protocol was modified to include the trace context.
- Trace logs were systematically written to a designated trace file. This trace file undergoes periodic rotation or truncation upon reaching a predefined size limit.

To minimize significant time discrepancies between events, peers periodically synchronize their clocks with Google NTP. However, this approach may not be sufficient to entirely eliminate clock skew, particularly in networks with geographically dispersed nodes. To address this challenge, the aggregation layer (described later) is responsible for adjusting any remaining clock skew based on causal relationships of the spans.
Trace files are constrained to a maximum size of 16MB (which was inspired by PostgreSQL's WAL).

*II. **HTTP Gateway Server***

We implemented an HTTP gateway server to handle request sampling and content publishing and retrieval. The global state of the gateway is stored in a database. Key functions of the gateway includes:

- *Health Checks***:** The gateway server periodically sends health check requests to each of the Nabu nodes. The health status information guides the server about which nodes to interact with for publishing and retrieving content.
- *Content Publishing*: The gateway server is responsible for forwarding the PUT request to one of the healthy Nabu peers.
- *Content Retrieval***:** The gateway randomly samples X% of all the CIDs for tracing purposes during content retrieval. The trace Id is included in the GET response headers from Nabu peer.

*III. **Log Exporter***

Our goal was to implement a lightweight framework on top of Nabu in a laissez-faire way. We implemented a daemon process to export trace logs. The Log Exporter is responsible for collecting the traces recorded by a Nabu peer and shipping them to a backend server for processing. The daemon process can accommodate multi-tenancy if multiple Nabu peers run on the same machine. Additionally, the daemon process also maintains a durable state in case of a system failure.

```
struct Span {
  int spanId, enum type,
  int startTime, int endTime,
  int nodeId, int peerNodeId,
  int parentSpanId, set<int> children
}
struct Trace { string traceId, Span[] spans }
map parentEvent {
  GET_PROVIDERS_SERVER: GET_PROVIDERS_CLIENT
  BITSWAP_SERVER: BITSWAP_CLIENT
}
map childEvent {
  GET_PROVIDERS_CLIENT: GET_PROVIDERS_SERVER
  BITSWAP_CLIENT: BITSWAP_SERVER
  BITSWAP_SERVER: READ_FROM_FILE_STORE
}
func processTrace(trace):
  for span in trace.spans:
    if parentEvent[span.type] is not NONE:
      parent = spans.find({
                        span.peerNodeId,
                        parentEvent[span.type] })
       if parent is not None:
          span.parent = parent.spanId
          parent.children.add(span.spanId)
        else:
          # Missing spans
    if childEvent[span.type] is not NONE:
        child = spans.find({
                        span.peerNodeId,
                        childEvent[span.type] })
        if child is not None:
          span.children.insert(child.spanId)
          child.parent = span.spanId
        else:
          # Missing spans
```

Figure 1: Simplified algorithm for establishing causal relationships between trace spans.

*IV. **Span Aggregator***

Lastly, we implemented a backend service responsible for collecting and aggregating the timestamp-based logs exported by the daemon process to build interval-based spans. The aggregated spans are stored in Jaegar [11] which is an open-source platform for processing and visualizing traces. It performs necessary span validation, as well as clock-skew adjustments based on the provided causal relationships. We used ElasticSearch [12] as the storage backend for our spans.

EVALUATION

How much resource and performance overhead does tracing add to the existing system?

Each request message incurs an additional 16 bytes of

overhead due to the inclusion of the trace context. Trace logs are written to the disk on a periodic basis. This avoids disk writes from the critical path. The ability of

the aggregation layer to handle missing logs ensures that only complete traces are displayed, mitigating the impact of potential data loss.

To evaluate how tracing affects GET request performance, we ran a private network with five peers on a single machine (8-core Intel i5). The experiment involved sending 1,000 pairs of GET requests for a specific CID, with every pair containing an untraced and traced request. Each request pair targeted a random node within the network. The experiment was repeated four times.

As shown in Table 2, tracing has minimal impact on the median latency. However, it can significantly impact tail latency, by up to 1.6x in processing. This is due to additional overhead introduced by propagating trace context across threads and the network. On an absolute scale, this impact should still be minimal as compared to the network latency between the peers.

Table 2: Latency observed with/without tracing.

Experiment	Traced?	Median (ms)	p99 (ms)	Max (ms)
1	*No*	7	38	72
	Yes	7.5	41	90
2	*No*	6	31	74
	Yes	6	38	87
3	*No*	7	49	128
	Yes	7	52	122
4	*No*	9	74	124
	Yes	9	88	196

EXPERIENCES

Setup: To assess the efficacy of the IPFS tracing mechanism, several modifications were necessitated within the Nabu implementation. These modifications aimed to isolate and amplify specific network behaviors for analysis purposes - route caching and optimistic BitSwap were disabled to force DHT lookups for every request, content caching was disabled to force network lookups for content retrieval, and flow control mechanisms for both lookup and retrieval requests were disabled to simplify analysis.

A private IPFS network was set up on Google Cloud, consisting of ten peers running on geographically dispersed, low-end *e2-small* virtual machines (VM). Due to Nabu's lack of native peer discovery functionality, the peers were statically connected within the network setup. Each peer was provided a unique node Id from 0 to 9.

For the HTTP gateway servers, high-end VMs were deployed across regions in the United States, Europe, and Asia. Google Cloud Firestore [13] served as the centralized repository for the gateway servers' global state. A load balancer was used to distribute incoming requests among these backend nodes, ensuring resilience against regional outages.

The gateway server transparently forwarded the PUT requests to Nabu peers using a round-robin policy. Additionally, it stored the received CIDs within its global state. Additionally, the gateway server offered a GET API to retrieve all content published on the private network. For each CID, a round-robin algorithm was employed to select a node for content retrieval, resulting in an even distribution of load across available Nabu peers. The server initiated each retrieval request concurrently to exert pressure on the peers. Upon receiving responses, the server streamed the retrieved content to the frontend along with supplementary information: the node that handled the request, the response time, and, if applicable, a link to the associated trace (for traced requests).

On every VM where a peer runs, we deployed the log exporter as a high-priority daemon with access to the trace logs.

The span aggregator backend service, responsible for aggregation, utilizes a primary-backup replication architecture. This architecture leverages a load balancer to facilitate healthy backend selection, ensuring that all logs are consistently delivered to a single backend instance for aggregation purposes.

The setup is illustrated in Figure 2.

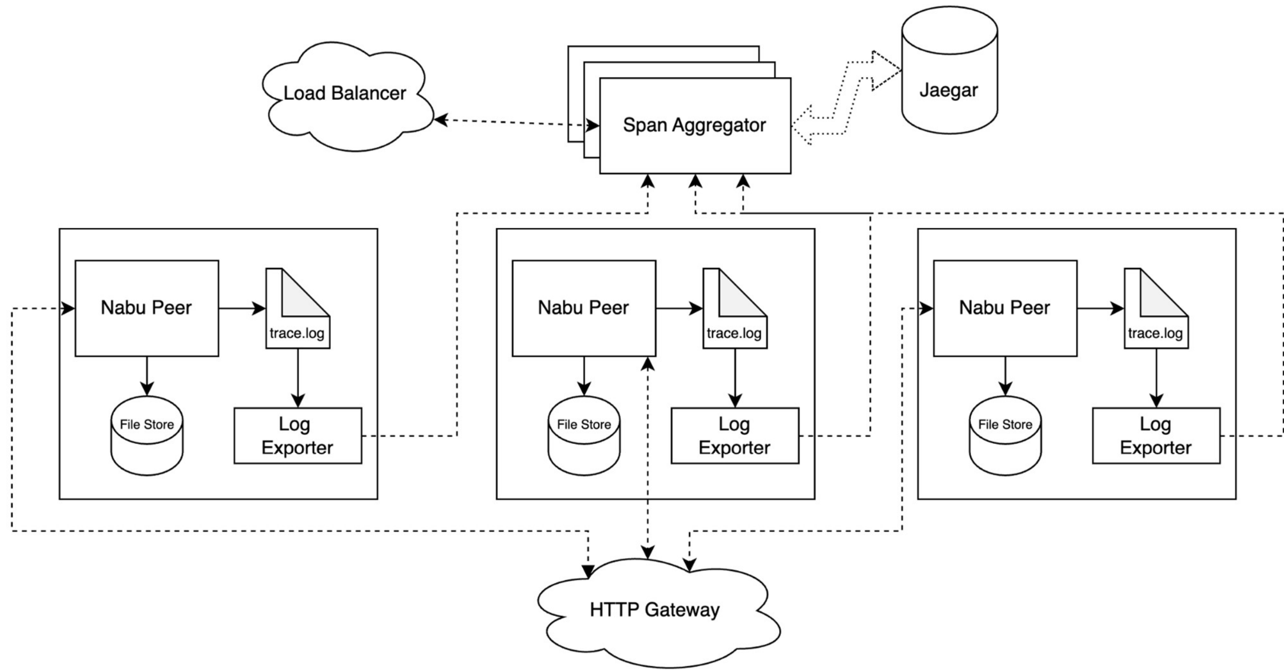

Figure 2: Tracing infrastructure setup on Google Cloud.

Results: Figure 3 presents the results for a sample trace collected for a request with a complex DHT lookup path and BitSwap execution.

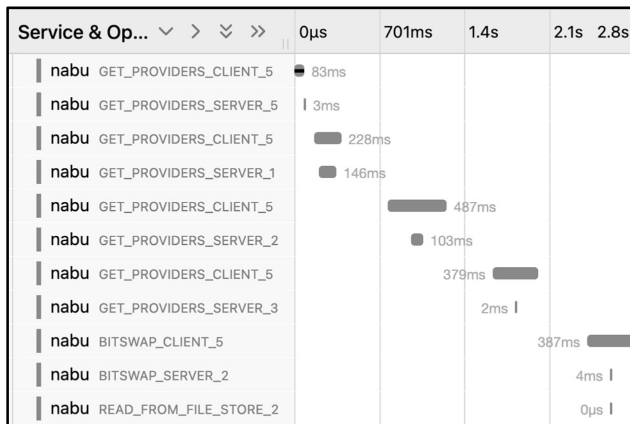

***Figure 3**:* Trace timeline representing the trace for a sample request. The request was initially directed towards node 5. Subsequently, node 5 initiated a DHT lookup sequence, starting with itself and continuing on to node 1, node 2, and finally node 3. Then, a BitSwap operation was performed with node 2. The network RTT to node 1 (co-located in Southeast Asia), exhibited low latency (~80ms). Conversely, the network RTT to node 2 and 3, both located in Europe, demonstrated significantly higher latency (~380ms).

To evaluate the effectiveness of our method in identifying slow nodes, a computationally intensive vector multiplication task was initiated on a designated node, consuming ~200% vCPU. A request was directed towards this node. The response for this request was received after a significant delay of 16.22s which is 5x of normal response time. Figure 4 shows the corresponding trace.

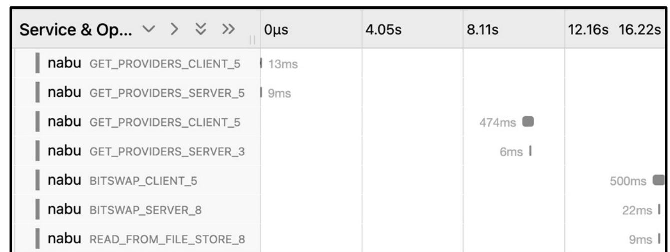

Figure 4: Trace timeline representing a request processing at node 5 while another process was running a heavy vector multiplication. The first DHT lookup finished in 13ms. Subsequent lookups and BitSwap experienced significant start delays of 8s and 6s respectively.

SECURITY CONSIDERATION

A fundamental challenge in decentralized systems lies in the vulnerability to malicious behavior of peers. For instance, a peer may log traces without a corresponding

event. By leveraging semantics to establish relationships between spans within traces, we are able to avoid collecting incomplete traces. Thus, our system would only show complete traces.

Future Work

The proposed tracing framework can be integrated with Kubo [14], the production Go implementation of IPFS. Furthermore, tracing capabilities could be incorporated into IPFS gateways [15].

The scope of traced events could be expanded to encompass additional operations, such as optimistic BitSwap, cache retrieval, and cached route lookups. Additionally, the breakdown of network latencies could be refined to include granular subevents, such as TCP buffer delays on both the sender and receiver sides.

Alternative sampling policies can also be explored to enhance the effectiveness of our distributed tracing system. Specifically, tail-based sampling [16] could be used to capture traces for requests that have high latency or result in errors. Adaptive sampling, as described in Dapper [4], may also be implemented to adjust sampling rate based on system load to reduce trace overhead.

Our current approach consists of a centralized entity collecting, storing, and summarizing traces. In the future, we would like to implement the capability for any participant in the decentralized network to collect and aggregate spans from our tracing system.

Conclusion

In this work, we presented a prototype implementation of a distributed tracing framework for the IPFS. This framework has the potential to be a valuable tool for a variety of users, particularly web owners. By enabling the tracing of requests within the IPFS network, the framework can provide detailed insights into the performance characteristics of content retrieval operations. Through analysis of traced data, web owners can identify bottlenecks and inefficiencies, allowing them to optimize content delivery and ensure a smooth user experience.

Acknowledgment

We are particularly grateful to David Mazières for his invaluable suggestion that led to the conception of this project. We would also like to extend our sincere thanks to Ian Preston for his work in developing Nabu, which served as the foundation upon which we built the tracing layer.

References

[1] Benet, J, July 2014. "IPFS - Content Addressed, Versioned, P2P File System." *Networking and Internet Architecture (cs.NI); Distributed, Parallel, and Cluster Computing*3. https://arxiv.org/abs/1407.3561

[2] Maymounkov, P., Mazières, D., Kademlia: A Peer-to-peer Information System Based on the XOR Metric. Revised Papers from the First International Workshop on Peer-to-Peer Systems, April 2002.

[3] Roacha, A., Dias, D., Psaras, Y., Accelerating Content Routing with Bitswap: A multi-path file transfer protocol in IPFS and Filecoin, 2021.

[4] Sigelman, B., Barroso, L., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C., Dapper, a large-scale distributed systems tracing infrastructure. Technical Report dapper-2010-1, Google, April 2010.

[5] Cloudera HTrace. http://github.com/cloudera/htrace

[6] Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neil, J., Win Ong, K., Schaller, B., Shan, P., Viscom, B.i, Venkataraman, V., Veeraraghavan, K., Jiun Song, Y., Canopy: An End-to-End Performance Tracing and Analysis System. Symposium on Operating Systems Principles (SOSP), October 2017.

[7] OpenTelemetry. https://opentelemetry.io/

[8] Fonseca, R., Porter, G., Katz, R., Shenker, S., and Stoica., I., X-Trace: a pervasive network tracing framework. In NSDI 07: Proceedings of the ¦th USENIX Symposium on Networked Systems Design and Implementation, 2007.

[9] Barham, P., Donnelly, A., Isaacs, R., and Mortimer,, R. Using Magpie for Request Extraction and Workload Modeling. In Proc. USENIX OSDI (2004).

[10] Preston, I., Introducing Nabu: Unleashing IPFS on the JVM. https://blog.ipfs.tech/2023-11-introducing-nabu. Web. Accessed: November 7, 2023.

[11] Jaeger. https://www.jaegertracing.io/

[12] ElasticSearch. https://www.elastic.co/guide/index.html

[13] Kesavan, R., Gay, D., Thevessen, D., Shah, J., Mohan, C., Firestore: The NoSQL Serverless Database for the Application Developer. 2023 IEEE 39th International Conference on Data Engineering (ICDE), pp. 3367-3379

[14] Kubo. https://github.com/ipfs/kubo

[15] IPFS Gateway. https://docs.ipfs.tech/concepts/ipfs-gateway/

[16] Lee, R. Tail Sampling with OpenTelemetry: Why it's useful, how to do it, and what to consider. https://opentelemetry.io/blog/2022/tail-sampling/.