End-to-end Tracing in HDFS

William Wang

July 2011
CMU-CS-11-120

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Greg Ganger, chair
David R. O'Hallaron

*Submitted in partial fulfillment of the requirements
for the Degree of Master of Science*

Keywords: computer science, HDFS, end-to-end tracing, Hadoop, performance diagnosis

Abstract

Debugging performance problems in distributed systems is difficult. Thus many debugging tools are being developed to aid diagnosis. Many of the most interesting new tools require information from end-to-end tracing in order to perform their analysis. This paper describes the development of an end-to-end tracing framework for the Hadoop Distributed File System. The approach to instrumentation in this implementation differs from previous ones as it focuses on detailed low-level instrumentation. Such instrumentation encounters the problems of large request flow graphs and a large number of different kinds of graphs, impeding the effectiveness of the diagnosis tools that use them. This report describes how to instrument at a fine granularity and explain techniques to handle the resulting challenges. The current implementation is evaluated in terms of performance, scalability, the data the instrumentation generates, and its ability to be used to solve performance problems.

1 Introduction

Diagnosing performance problems in distributed systems is hard. Such problems often have multiple sources, could be contained in any number of components within in the system, and even could be the result of the interactions between components. For this reason, there is ongoing development of debugging tools for guiding perfomance problem diagnosis.

In particular, many build upon end-to-end tracing frameworks. In end-to-end tracing, one captures information about the flows (paths and timings) that individual requests take to be serviced in the distributed system. In this way, end-to-end tracing is able to capture the behavior of each of the system's components and also the interactions between them. As one example of its utility, tools can use this information to characterize the normal behavior of the system and detect the changes in behavior during periods of poor performance [12]. One can look at these changes to narrow down and find the root causes of the decreased performance.

Apache Hadoop [10] is an open source implementation of Google's MapReduce. It is composed of two main components. One component is the MapReduce software framework for the distributed processing of large data sets on compute clusters. The other is the Hadoop Distributed File System (HDFS), a distributed file system that provides high throughput access and is commonly used by the MapReduce component as its underlying file system for retrieving input and outputting results. As with any other distributed system, performance problem diagnosis is difficult in Hadoop. Due to the utility of end-to-end tracing, this thesis project explored how to implement it in HDFS. In specific, the contributions of this project are:

1. Design and implementation of end-to-end tracing framework for HDFS

2. Identification of places where instrumentation should be added for effective end-to-end tracing in HDFS

3. Design of a robust and reliable collection backend

4. Evaluation of the scalability and performance of the framework.

5. Graphs of the behavior of HDFS's filesystem calls. These are useful in helping developers build intuition about HDFS's internal structure.

6. Preliminary work showing how end-to-end traces collected from HDFS can be used in existing debugging tools

7. Preliminary design of a querying API for accessing raw trace records.

This report is structured as follows. Section 2 provides a definition of end-to-end tracing and presents several tools that use it. Section 3 gives a background on Hadoop and HDFS, including previous approaches of tracing in this framework. Section 4 presents the goals of the end-to-end tracing framework and resulting challenges. Section 5 shows the approach to implementing the framework. Section 6 presents the design of the collection and storage services in the framework as well the as API for accessing debugging information. Section 7 discusses the implmentation of the instrumentation in HDFS. Section 8 evaluates various characteristics of the framework, such as its scalablility, the size of the data generated by the framework, and its ability to be used by diagnosis tools to diagnosis problems. Some future work related to the framework is discussed in section 9, and we conclude in section 10.

2 End-to-end Tracing

End-to-end tracing captures the flow of requests. This is achieved by capturing activity records at the various instrumentation points placed in the distributed system components. Each record contains information such as instrumentation point name, a timestamp, and optionally other contexual information. In general, records are associated with individual requests and thus also propagate a request identifier. These records can then be stitched together to form a request flow graph that shows the control flow (see Figure 1). In such a graph, nodes are labeled by the instrumentation point name and edges are labeled with the latency that the request experienced between one instrumentation point and the next. As noted previously, end-to-end tracing can simplify the process of performance problem diagnosis with the rich information it provides. Such tracing can be implemented with low overhead as seen by the multiple independent implementations, such as Dapper [13], Magpie [2], Stardust [14], or X-Trace [7]. This section discusses several end-to-end tracing frameworks and diagnosis tools.

2.1 End-to-end Tracing Frameworks

There are several end-to-end tracing frameworks that have been implemented and have some different characteristics. Dapper [13] is Google's end-to-end tracing framework. Unlike other frameworks, Dapper instruments RPCs as single units, and thus provides a much higher level view. It also does not explicitly capture sequential and parallel activity. Magpie [2] does not propagate request identifiers and requires the developer to input a schema that defines the structure of the request flow graph. Stardust and X-Trace are two other frameworks which are very similar. Both propagate request identifiers and both explicity capture sequential and parallel activity.

2.2 Spectroscope

The purpose of adding end-to-end tracing support to HDFS is to enable use of tools which analyze request flow graphs. One such tool is Spectroscope [12]. Spectroscope is a tool for comparing request flows across periods in order to find the root causes of behaviorial differences between them. For performance diagnosis, it would be used to compare the request flows of a non-problem period, a period in which performance of the system is within expectations, and a problem period, a period in which performance is worse than expected. It does this by first organizing the graphs from both periods into categories of identical structure. For example, all the **`mkdir`** requests from the same client would be part of the same category in HDFS. Then, it compares the graphs between the two periods in order to find mutations or changes in the graph.

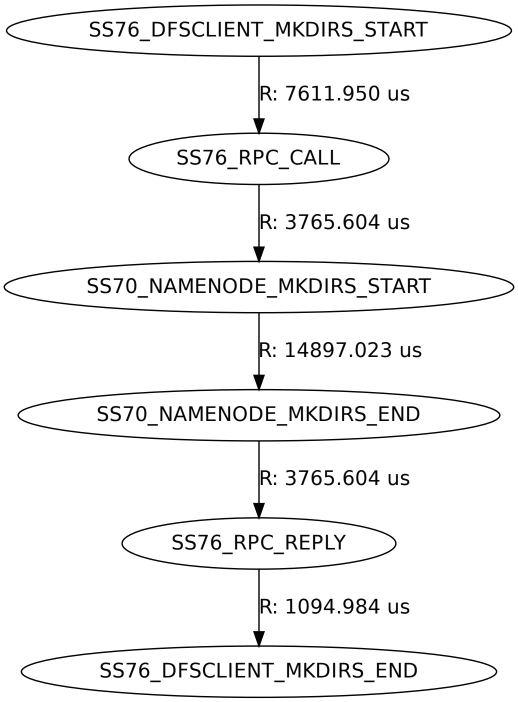


Figure 1: Example request flow graph of a **`mkdirs`** RPC in HDFS. A client made a request to create a new directory and optional subdirectories in the HDFS instance. As a result of the call, an RPC was issued and sent to the namenode, the master node in HDFS. The namenode did the required processing and returned control to the client.

Spectroscope looks at two types of mutations. The first type is the response-time mutation. In this kind of mutation, the latencies on edges in a graph category in one period are significantly different from the same category of graphs in the other period. These kinds of mutations are found by performing a statistical hypothesis test. In the test, the null hypothesis is that the samples from the two periods are from the same distribution. If the null hypothesis is rejected, then a response-time mutation was detected. One may see this kind of mutation in cases such as a bad network link. The communication latency would increase between components, which would manifest itself in the graph as a edge with an increased latency.

The other kind of mutation is the structural mutation. This kind of mutation is the result of a request taking a different path than was expected. This results in the associated request flow graph taking on a different structure. Spectroscope finds these kinds of mutations by comparing across categories and attempting to find the graph category that the mutated graph originated from. This kind of mutation can be seen in requests that access caches.

After finding mutations, Spectroscope then ranks the mutations by their expected contribution to the performance change. In this way, developers know where to focus their diagnosis effort.

Spectroscope has been shown to be useful in diagnosing performance problems in systems such as Ursa Minor [1] and certain Google services. It is likely that HDFS would benefit from enabling the use of Spectroscope.

2.3 Other Tools

There also exist other tools that attempt to detect anomalous behavior within a single period as opposed to comparing behaviors between two periods like Spectroscope, such as Magpie [2] and Pinpoint [4]. Magpie using behaviorial clustering in order to sort requests into clusters. By creating clusters, Magpie performs anomaly detection by noting the requests that are outliers and do not fit into any cluster. Pinpoint uses statistical machine learning algorithms to attempt to model normal path behavior by calculating the likelihood of a path occurring based on training data. Paths that are unlikely to occur are anomalous and can be used to pinpoint the root cause of the problem. In this report, we focus on Spectroscope.

3 Hadoop Distributed File System

HDFS is the open source implementation of the Google File System [8] (GFS). It is a distributed file system created with the purpose of storing large data for large scale data intensive applications. Like GFS, the architecture of HDFS consists of a single master, called the namenode, and multiple storage servers, called datanodes. Files are divided into fixed size blocks which are distributed among the datanodes and stored on the datanodes' local disks. Often, blocks are replicated across multiple datanodes for the purpose of reliability. The namenode stores all file system metadata information and file-to-block mappings and controls system-wide activities such as lease management, garbage collection, and block migrations between datanodes.

As with most file systems, two common operations in HDFS are reading and writing files. To read a file, a client requests the locations of the blocks of the file on the datanodes from the namenode. After receiving the block locations, the client then requests block transfers from datanodes that have the blocks in order to read the file.

To write a file, a client requests datanodes from the namenode that it should send blocks to. If replication is activated in the HDFS instance, a set of datanodes is given to a client for each block. In HDFS, blocks are generally 64MB in size by default and are transferred between components in 64KB checksumed packets[1], leading to over one thousand HDFS packet transfers for a single block. To improve the performance and decrease the latency of replicated writes, the client and datanodes are formed into a pipeline. The pipeline is usually constructed such that the "distance" between successive nodes is minimized to avoid network bottlenecks and high-latency links as much possible. However, the namenode is allowed to break this requirement and arbitrarily choose the datanodes and their ordering to satisfy other constraints. The client only sends packets to the first datanode in the pipeline. While doing so, it puts the packet into a queue that asynchronously waits for acknowledgements. When a datanode receives a packet, it sends the packet to the next

[1] Packet refers to an HDFS packet, which is made of multiple IP packets

datanode in the pipeline unless it is the last datanode in the pipeline in which case it will send an acknowledgement to the previous datanode. When a datanode receives an acknowledgement from the next datanode, it will also send an acknowledgement to the previous datanode or client. When the client receives the acknowledgement from the first datanode, it knows that all the datanodes in the pipeline have successfully received the packet and thus the packet can be removed from the queue.

HDFS is often used with MapReduce applications to store input and output. In many kinds of MapReduce jobs, a significant amout of time is spent reading the input data and writing the output data. Thus, the performance of HDFS can directly affect the performance of jobs. For this reason, it is valuable to look into ways to diagnosis performance problems in HDFS.

4 Goals and Resulting Challenges

This section describes the goals for the HDFS end-to-end tracing framework. We encountered several challenges as a result of the chosen goals, which will also be discussed in this section.

4.1 Goals

4.1.1 Low overhead

Ideally, an end-to-end tracing framework should always be running, even in a production system. This is so that, if a performance problem were to be detected, one can immediately look at the resulting traces as they have already been collected, which then speeds up the diagnosis process. Otherwise, the process will be much slower as it would require tracing to be enabled, traces to be collected, and then tracing to be disabled. In doing so, one may have to wait until the problem appears again, which could happen again in some arbitrary time in the future. However, always-on tracing is only achievable if the framework incurs a low performance overhead since few are willing sacrifice performance in their systems.

4.1.2 Detailed Low-level Instrumentation

One of goals is to instrument at a fine granularity. For example, we have instrumented at the HDFS packet level for the writes described in section 3. The reason for this is to expose as much detail as possible to the developer. Also, in this way, tools such as Spectroscope can be even more precise about the locations of the mutations in a request category, possibly helping to pinpoint the root cause faster and more precisely. Another benefit of low-level instrumentation is its ability for its traces to be aggregated to create a higher-level view if necessary. Essentially, by instrumenting at the lowest level, we can expose information at any level above it. Thus, such a framework can be used to capture problems both at the micro and macro level.

4.2 Challenges

4.2.1 Large Graphs

As a result of the detailed instrumentation, some graphs are very large. This occurs in HDFS when capturing the behavior during the execution of a HDFS block write request. Figure 2 shows a example of such a request flow graph. This graph is the result of capturing every packet send iteration, each of which asynchronously waits for an acknowledgement. Depending on how many iterations the operation must perform to complete the task, the graph can grow very large very quickly. In the case of HDFS, it is dependent on the replication factor. However in all cases, the number of nodes in a full block write is on the order of thousands. This is a problem for diagnosis tools as they are generally not written to efficiently handle graphs of that size. Some tools will run out of memory attempting to interpret the graph. GraphViz [9], a graph visualizer, is one such tool. Spectroscope spends hours analyzing small workloads that generate small numbers of large graphs. This implies that the effect of large graphs is very significant. Also, large graphs make it difficult to achieve the goal of low overhead as large numbers of activity records will be captured even in small workloads, thus likely affecting the performance of the instrumented system due to the extra network traffic and processing.

4.2.2 Category Count Explosion

As a result of replication and the fact that the namenode is allowed to choose any set and any ordering of datanodes, there can be a very large number of categories for writes. Given n cluster nodes and a replication factor of r, there can be ${}_nP_r \in \Theta(n^r)$[2] unique sets of datanodes and orderings. Also, there also are different graph structures based on amount of data written. In general, these graphs must be in separate categories as the request physically took different paths. The effectiveness of a tool such as Spectroscope is weakened as a result of a large number of categories because the number of samples per category will be small, reducing power of the statistical comparison.

5 Approach

The general approach that was taken to implement the end-to-end tracing framework for HDFS was to use existing building blocks as a foundation and modify as needed. The X-Trace [7] framework was chosen as the primary foundation. This section describes the X-Trace framework, the reasons for using it, and the challeges which emerged as the result of this choice.

5.1 X-Trace

X-Trace is a complete end-to-end tracing framework that can be installed into any system to enable end-to-end tracing. In X-Trace's instrumentation API, the activity record unit is called a report. It has the form as shown below

[2] ${}_nP_r$ is bounded above and below by polynomials in n of degree r. ${}_nP_r$ is the number of permutations of r elements that can be made from n elements

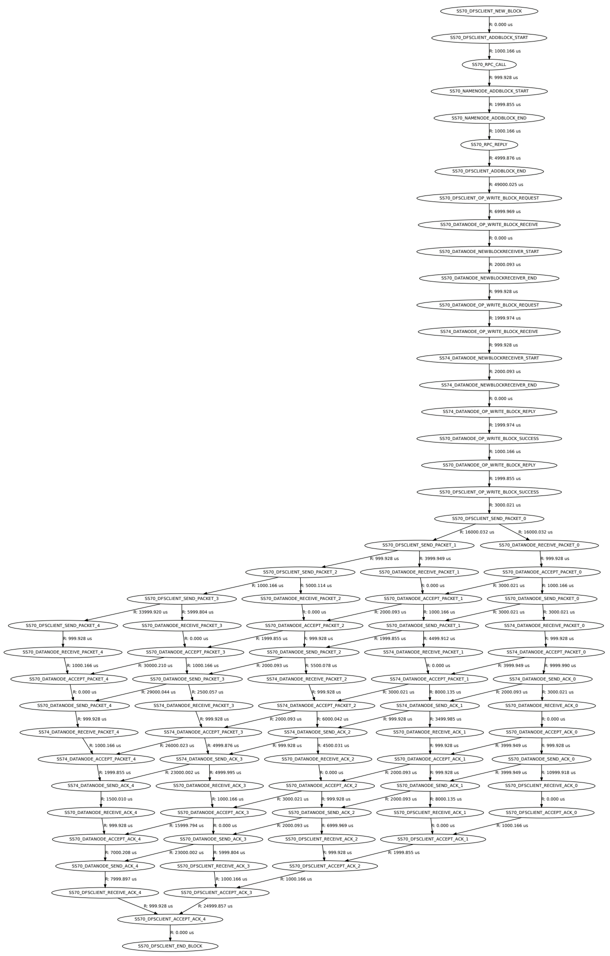

SS70_DFSCLIENT_NEW_BLOCK
R: 0.000 us
SS70_DFSCLIENT_ADDBLOCK_START
R: 1000.166 us
SS70_RPC_CALL
R: 999.928 us
SS70_NAMENODE_ADDBLOCK_START
R: 1999.855 us
SS70_NAMENODE_ADDBLOCK_END
R: 1000.166 us
SS70_RPC_REPLY
R: 4999.876 us
SS70_DFSCLIENT_ADDBLOCK_END
R: 49000.025 us
SS70_DFSCLIENT_OP_WRITE_BLOCK_REQUEST
R: 6999.969 us
SS70_DATANODE_OP_WRITE_BLOCK_RECEIVE
R: 0.000 us
SS70_DATANODE_NEWBLOCKRECEIVER_START
R: 2000.093 us
SS70_DATANODE_NEWBLOCKRECEIVER_END
R: 999.928 us
SS70_DATANODE_OP_WRITE_BLOCK_REQUEST
R: 1999.974 us
SS74_DATANODE_OP_WRITE_BLOCK_RECEIVE
R: 999.928 us
SS74_DATANODE_NEWBLOCKRECEIVER_START
R: 2000.093 us
SS74_DATANODE_NEWBLOCKRECEIVER_END
R: 0.000 us
SS74_DATANODE_OP_WRITE_BLOCK_REPLY
R: 1999.974 us
SS70_DATANODE_OP_WRITE_BLOCK_SUCCESS
R: 1000.166 us
SS70_DATANODE_OP_WRITE_BLOCK_REPLY
R: 1999.855 us
SS70_DFSCLIENT_OP_WRITE_BLOCK_SUCCESS
R: 3000.021 us
SS70_DFSCLIENT_SEND_PACKET_0
R: 16000.032 us
R: 16000.032 us
SS70_DFSCLIENT_SEND_PACKET_1
SS70_DATANODE_RECEIVE_PACKET_0
R: 999.928 us
R: 3999.949 us
R: 999.928 us
SS70_DFSCLIENT_SEND_PACKET_2
SS70_DATANODE_RECEIVE_PACKET_1
SS70_DATANODE_ACCEPT_PACKET_0
R: 1000.166 us
R: 5000.114 us
R: 0.000 us
R: 3000.021 us
R: 1000.166 us
SS70_DFSCLIENT_SEND_PACKET_3
SS70_DATANODE_RECEIVE_PACKET_2
SS70_DATANODE_ACCEPT_PACKET_1
SS70_DATANODE_SEND_PACKET_0
R: 33999.920 us
R: 5999.804 us
R: 0.000 us
R: 2000.093 us
R: 1000.166 us
R: 3000.021 us
R: 3000.021 us
SS70_DFSCLIENT_SEND_PACKET_4
SS70_DATANODE_RECEIVE_PACKET_3
SS70_DATANODE_ACCEPT_PACKET_2
SS70_DATANODE_SEND_PACKET_1
SS74_DATANODE_RECEIVE_PACKET_0
R: 999.928 us
R: 0.000 us
R: 1999.855 us
R: 999.928 us
R: 1999.855 us
R: 4499.912 us
R: 999.928 us
SS70_DATANODE_RECEIVE_PACKET_4
SS70_DATANODE_ACCEPT_PACKET_3
SS70_DATANODE_SEND_PACKET_2
SS74_DATANODE_RECEIVE_PACKET_1
SS74_DATANODE_ACCEPT_PACKET_0
R: 1000.166 us
R: 30000.210 us
R: 1000.166 us
R: 2000.093 us
R: 5500.078 us
R: 0.000 us
R: 3999.949 us
R: 9999.990 us
SS70_DATANODE_ACCEPT_PACKET_4
SS70_DATANODE_SEND_PACKET_3
SS74_DATANODE_RECEIVE_PACKET_2
SS74_DATANODE_ACCEPT_PACKET_1
SS74_DATANODE_SEND_ACK_0
R: 0.000 us
R: 29000.044 us
R: 2500.057 us
R: 999.928 us
R: 3000.021 us
R: 8000.135 us
R: 2000.093 us
R: 3000.021 us
SS70_DATANODE_SEND_PACKET_4
SS74_DATANODE_RECEIVE_PACKET_3
SS74_DATANODE_ACCEPT_PACKET_2
SS74_DATANODE_SEND_ACK_1
SS70_DATANODE_RECEIVE_ACK_0
R: 999.928 us
R: 999.928 us
R: 2000.093 us
R: 6000.042 us
R: 999.928 us
R: 3499.985 us
R: 0.000 us
SS74_DATANODE_RECEIVE_PACKET_4
SS74_DATANODE_ACCEPT_PACKET_3
SS74_DATANODE_SEND_ACK_2
SS70_DATANODE_RECEIVE_ACK_1
SS70_DATANODE_ACCEPT_ACK_0
R: 1000.166 us
R: 26000.023 us
R: 4999.876 us
R: 999.928 us
R: 4500.031 us
R: 999.928 us
R: 3999.949 us
R: 999.928 us
SS74_DATANODE_ACCEPT_PACKET_4
SS74_DATANODE_SEND_ACK_3
SS70_DATANODE_RECEIVE_ACK_2
SS70_DATANODE_ACCEPT_ACK_1
SS70_DATANODE_SEND_ACK_0
R: 1999.855 us
R: 23000.002 us
R: 4999.995 us
R: 0.000 us
R: 2000.093 us
R: 999.928 us
R: 3999.949 us
R: 10999.918 us
SS74_DATANODE_SEND_ACK_4
SS70_DATANODE_RECEIVE_ACK_3
SS70_DATANODE_ACCEPT_ACK_2
SS70_DATANODE_SEND_ACK_1
SS70_DFSCLIENT_RECEIVE_ACK_0
R: 1500.010 us
R: 1000.166 us
R: 3000.021 us
R: 999.928 us
R: 2000.093 us
R: 8000.135 us
R: 0.000 us
SS70_DATANODE_RECEIVE_ACK_4
SS70_DATANODE_ACCEPT_ACK_3
SS70_DATANODE_SEND_ACK_2
SS70_DFSCLIENT_RECEIVE_ACK_1
SS70_DFSCLIENT_ACCEPT_ACK_0
R: 999.928 us
R: 15999.794 us
R: 0.000 us
R: 2000.093 us
R: 6999.969 us
R: 0.000 us
R: 1000.166 us
SS70_DATANODE_ACCEPT_ACK_4
SS70_DATANODE_SEND_ACK_3
SS70_DFSCLIENT_RECEIVE_ACK_2
SS70_DFSCLIENT_ACCEPT_ACK_1
R: 7000.208 us
R: 23000.002 us
R: 5999.804 us
R: 999.928 us
R: 1999.855 us
SS70_DATANODE_SEND_ACK_4
SS70_DFSCLIENT_RECEIVE_ACK_3
SS70_DFSCLIENT_ACCEPT_ACK_2
R: 7999.897 us
R: 1000.166 us
R: 1000.166 us
SS70_DFSCLIENT_RECEIVE_ACK_4
SS70_DFSCLIENT_ACCEPT_ACK_3
R: 999.928 us
R: 24999.857 us
SS70_DFSCLIENT_ACCEPT_ACK_4
R: 0.000 us
SS70_DFSCLIENT_END_BLOCK

Figure 2: A request flow graph for a the write of a partial block with a replication factor 2 and 5 packet sends. As an HDFS packet is being sent through the pipeline, another packet can be sent without waiting for the acknowledgement for the previous packet. This leads to a fork in the graph for each packet send and the diamond shape seen in the graph, where each column represents a single packet being sent through the pipeline.

```
X-Trace Report ver 1.0
X-Trace: 194C79295549A3866ECCC315DD47301EA7
Host: ww2-laptop
Agent: Namenode
Label: WW2-LAPTOP_NAMENODE_MKDIRS_START
Edge: E7EB8787515DA35E
Timestamp: 181854865781004
```

Essentially, it is a collection of key-value pairs of the form $key : value$. There are several required keys, one of which is **X-Trace**, the unique identifier of the report. Its value is composed of three parts. The first byte describes the length of the next two components of the metadata using X-

Trace's defined bit representation. 19 means that they are both 8 bytes in length. The task ID is the next component and is follwed by the event ID. These two components are called the report's metadata. The task ID is a random number assigned to a request and is constant across all reports associated with the same request. The event ID is a unique number used to differentiate reports of the same request. Reports are stitched together to form a request flow graph using the **Edge** key. The value of an **Edge** is the event ID of a parent report. For instrumentation points to recognize a parent, the metadata of the last report must be known. Within a component, this is done by saving the metadata either as a thread local variable in a thread-based system or as part of a data structure in case of a event-based system. In the first case, we use the **setThreadContext()** method to set the thread local variable. In the second case, we save the value retrieved from **getThreadContext()** in the data structure used in callbacks. Between components, this is achieved by augmenting communication protocols of the system to send X-Trace metadata between instrumented components. Here is an example of adding the metadata to a protocol:

```
...
byte[] buf =
  XTraceContext.getThreadContext().pack();
out.writeInt(buf.length);
out.write(buf);
...
out.flush()
```

Simiarly, on the other side, we insert some code to retrieve the metadata:

```
...
int len = in.readInt();
byte[] bf = new byte[len];
in.readFully(bf);
XTraceMetadata meta =
 XTraceMetadata.createFromBytes(bf, 0, len);
XTraceContext.setThreadContext(meta);
...
```

There were several reasons we chose X-Trace. The metadata that is required by X-Trace is small and constant size, which allows X-Trace to achieve a low instrumentation overhead. There had been previous work in instrumenting Hadoop and HDFS using X-Trace [15]. According to the homepage, Hadoop and HDFS had been completely instrumented, and they could produce useful graphs that could be used to aid in diagnosis. This project is a proof of concept that X-Trace could successfully be used to instrument Hadoop and produce useful results.

5.1.1 Architecture

Figure 3 shows the architecture of the X-Trace framework. The instrumented software components send their records to a local port. In HDFS, the instrumented components would be the clients, namenode, and datanodes. An X-Trace proxy daemon runs on the same node as the instrumented component, reads from that local port and forwards the records to the collection server. The collection server then stores the records in a local database, which sorts them by task ID. Using a web interface, one could query the database for specific request flow graphs.

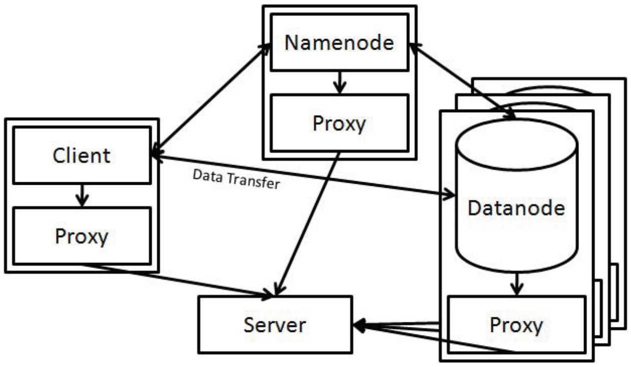


Figure 3: X-Trace architecture: This figure shows how X-Trace would be integrated in an HDFS cluster. Every node runs a proxy, which would collect X-Trace reports locally and then forward them to a central server.

5.2 Poor scalability of X-Trace

There were two issues we encountered when attempting to use the unmodified X-Trace framework:

- X-Trace was not robust to activity record drops. As described in section 2, the stitching mechanism in X-Trace relies on an **`Edge`** key inserted in the report that would contain the value of the parent's event ID. However, this means that if any record were dropped, some node's parent will not exist and the graph will not be connected. We lose the total ordering of the records, and the graph is no longer meaningful. As the scale of the cluster increases, the likelihood of record drops also increase, which implies an increase in request flow graphs that cannot be constructed, which will decrease the utility of diagnosis tools as they have less information to use.

- X-Trace's central server is not scalable. The server dumps all records into a shared bounded buffer that a separate thread asynchronously empties and inserts into the database. Because of our detailed instrumentation, we generate activity records at a rate much faster than the buffer empties and, thus, the buffer overflows. This, in combination with the above problem, leads to a high probability that a request flow graph be unable to be stitched together. This problem is only exacerbated as the number of nodes in the cluster increases.

The two charts in figure 4 show two different measurements of scalability. The first shows the percentage of malformed graphs formed during a sort job as cluster size increases. A malformed graph is any graph that is detected to be missing some activity record in the process of being constructed. Thus, we can view the number of malformed graphs as a measurement of activity records that were dropped end-to-end. For these sets of experiments, the malformed graph detection algorithm was not strict, so the percentage may be an underestimate of the true value. The checks that were made were as follows:

- The root node's label matched the last node's label. E.g., if the root had label **`FUNCTION_START`**, the last node must have label **`FUNCTION_END`**.
- The graph is connected.
- There is exactly one node with indegree 0, i.e. there is only one node that can be designated as root.
- There is exactly one node with outdegree 0, i.e. there is only one node that marks the end of the request.

Regardless, we can see approximately 10% of graphs being malformed, implying that the absolute number of malformed graph increases linearly as cluster size increases.

The second chart shows the percentage of activity records dropped according to the server, because of buffer overflow. The reason we also present this chart is because the "'percentage of malformed graphs measurement"' is an indirect measurement of record drops, which is often not a 1-to-1 relationship. A graph is considered malformed if it is missing at least one record. The graphs generated by these workloads vary greatly in size. Some have over 15000 nodes while others only have 6. Therefore, if activity records are dropped uniformly at random, it is quite likely that many of them were part of the same graph. Regardless of the number of nodes dropped from a single graph, it counts only as one malformed graph, which underrepresents the number of record drops. Another issue is that if all the activity records for a request flow graph were dropped, this graph is not reported at all in the malformed or total graph count. If we look figure 4(b), we can see that percentage more than tripled between 10 and 20 nodes, meaning that X-Trace does not scale at all passed 20 nodes.

6 Design

This section describes the various aspects of the current design of the framework. The original X-Trace framework did not scale well and thus we modified the architecture. The modifications are discussed in this section. Also, we will describe the preliminary API for accessing activity records, and we present approaches for dealing with the graph challenges.

6.1 New Architecture

There already exist stable implementations for each of components that are required for a scalable end-to-end tracing framework. The code of these open source software components have been tested and continually revised and improved and likely to be robust to failure. Thus, we constructed the end-to-end tracing framework by combining such components. In this way, we were able to quickly create a robust system where the only code that required extensive testing was the glue code that makes each of the components compatible with each other. This allows us to focus on improving the system's ability to gather information for performance diagnosis.

Figure 5 shows the new architecture. We replaced the X-Trace collection and storage components with Flume [5] and HBase [11] respectively, which we believe are more scalable than the original components.

6.1.1 Flume

Flume [5] is a distributed collection service developed by Cloudera. An instance of Flume contains a master and agent nodes. The master keeps track of the state of the instance. Each agent can be assigned a source, a location from which to receive log data, and a sink, a location to which to send log data, by the master. In the new architecture, we replaced the X-Trace proxy daemon with a Flume agent. We configure the master to assign the source to be the same local port as in the previous architecture. Instead of the X-Trace collection server, the sink is assigned to be the specific HBase table that will store the activity records.

6.1.2 HBase

HBase [11] is the open source implementation of BigTable [3]. Like BigTable, it is a distributed structured data storage system meant to scale to large sizes. It is essentially a multidimensional sorted map[3]. The map is indexed by row key, column key, and timestamp. The values stored in the map are uninterpreted arrays of bytes. Row keys are arbitrary strings and are used to lexigraphically sort the data. Columns are of the form *family:qualifier*. This is because column keys are grouped

[3] A data structure composed of a collection of unique keys and a collection of values where each key is associated with one value.

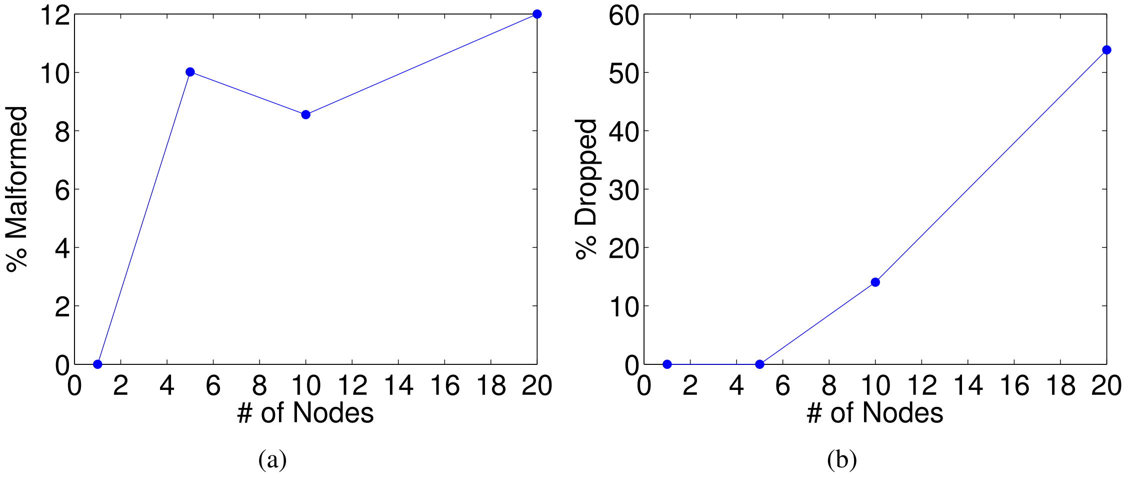


Figure 4: These charts measure the scalability of the X-Trace architecture. The first chart shows the percentage of requests whose graphs are malformed. The second chart shows the percentage of activity records dropped by the server because of buffer overflow.

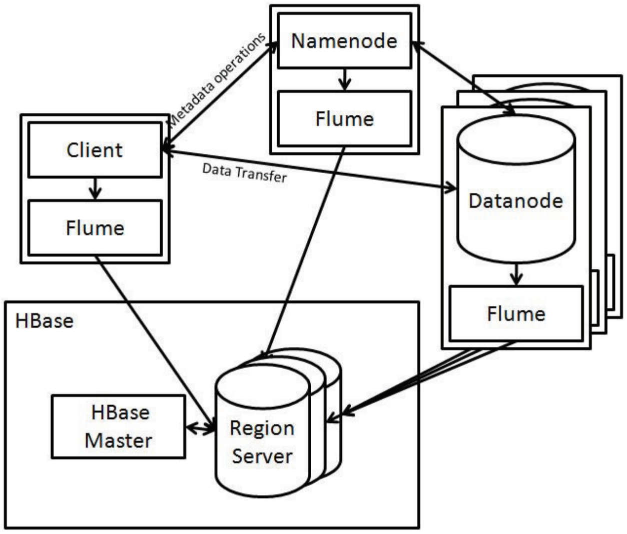


Figure 5: New X-Trace Architecture: The major changes to the architecture include replacing the proxy daemons with Flume nodes and replacing the central server with an HBase instance.

in sets called column families. Usually, data stored in a column family is of the same type (since all data in the same column family is compressed). Column families must be created before storing data under any column key of that family. After a family is created, any column key can be used within that family. We chose HBase as the storage backend for activity records in our new architecture, because it has been shown that the BigTable data model scales very well.

Schema Since the map is lexigraphically sorted based on row key, we chose a schema for naming rows and columns such that activity records that are closely related to each other will appear near each other in the map. The schema for the current implementation is relatively simple and is shown in Table1 with some example rows. The **`jTId`** is normally the ID of the map or reduce task in a mapreduce job. In the case that the request was not from a MapReduce job, the **`jTId`** is a random number. If a request is system initiated, the **`jTId`** is the string **`system`** concatenated with a random number. The **`taskID`** and **`eventID`** are the two parts of the X-Trace metadata. The **`pID`** is for stitching records together and represents the **`Edge`** key in an X-Trace report. Each row contains 1 column, which is the instrumentation point name and is part of the column family, **`inst_pt_name`**. The value in the column is any arbitrary value as it is unused. This is a simple schema that groups reports of the same job together. Then, they are further grouped by request. In this way, a scan of the map can efficiently get all the rows associated with a job or job and request, which will likely be the most common queries to this table.

6.1.3 API

We use a single base method for requesting traces from the table:

Row Key	Column Key
	inst_pt_name:FOO_START
<jTId> | <taskId> | <[pId,...]eventID> | <timestamp>	1
1|12|17|13245	1
attempt_201107062337_0015_m_001435_0|7|1|123	1
system12|19|123|1444	1

Table 1: Example rows in table: This table shows the format of the row key and column as well as some example entries of these three kinds (standalone, MapReduce, and system) of requests in HDFS.

getActivityRecords(jobId, taskId, start, end) This method will return all the activity records whose job ID matches **jobId** and whose task ID matches **taskId** between the times **start** and **end** in X-Trace report format. **jobId** and **taskId** can both be wildcard strings. Each of these parameter have default values and would cause the method to return all traces, (i.e. **jobId=``*''**, **taskId=``*''**, **start=0**, and **end=MAX_VALUE**, where **MAX_VALUE** is the maximum value of a long) such that, in general, not all of them have to be specified. The method is also defined such that it is independent of schema. The API was also designed such that it would be simple to get all the records of a particular workload (e.g., using jobId), and comparing them to a different workload. Also, limiting the times allows one to compare traces received during a non-problem period to traces in a problem period.

6.2 Graph Compression

To address very large graphs, we use compression of the graph structure. We attempt to reduce the number of nodes and edges that a graph contains while retaining the extra information in a different, simpler format. In many cases, large graphs are the result of large numbers of repeated subgraphs, which we see in HDFS block writes. Each HDFS packet that travels through the pipeline touches the same set of instrumentation points in the same order. Therefore, we can compress the graph by replacing the repeated copies of the subgraph with a single representative subgraph. In order to achieve this, treat the instrumentation points as states in a state machine. Moving from a instrumentation point to instrumentation point is a transition between states. Then, we can create a new graph where the nodes are the unique instrumentation points and edges are possible transitions between states. A subgraph in the original graph is isomorphic to a set of transitions that were made in the state machine. Then in the new graph, we allow edges to be labeled with a set of latencies, instead of only a single latency. For each edge in the subgraph, we find the corresponding transition in the new graph and add the latency of that edge to that transition's latency set. In this way, we have compressed the graph's structure while retaining the information in the original graph. Figure 6 shows an example of a write graph after it has been compressed.

The described compression algorithm can also slightly reduce the number of categories. This is because we would compress a full block write and partial block into the same kind of graph as the difference between them is the number of repeated subgraphs. However, this does not affect the number of possible permutations, which is still a problem. Thus, we also want compress the

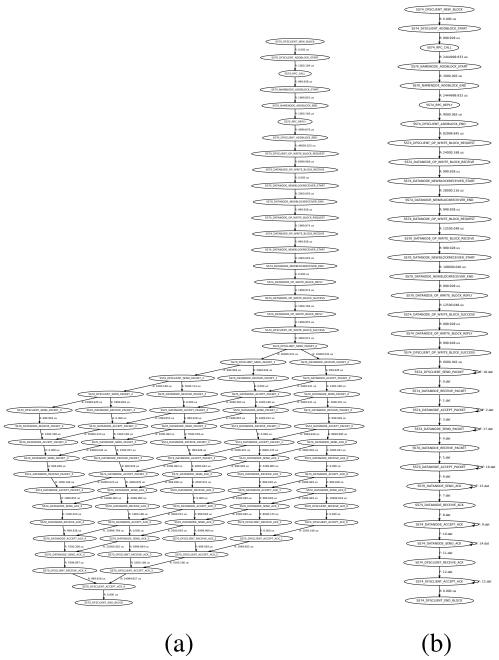

SS74_DFSCLIENT_NEW_BLOCK
R: 0.000 us
SS74_DFSCLIENT_ADDBLOCK_START
R: 999.928 us
SS74_RPC_CALL
R: 2444999.933 us
SS70_NAMENODE_ADDBLOCK_START
R: 2000.093 us
SS70_NAMENODE_ADDBLOCK_END
R: 2444999.933 us
SS74_RPC_REPLY
R: 9000.063 us
SS74_DFSCLIENT_ADDBLOCK_END
R: 82999.945 us
SS74_DFSCLIENT_OP_WRITE_BLOCK_REQUEST
R: 24000.168 us
SS74_DATANODE_OP_WRITE_BLOCK_RECEIVE
R: 999.928 us
SS74_DATANODE_NEWBLOCKRECEIVER_START
R: 28000.116 us
SS74_DATANODE_NEWBLOCKRECEIVER_END
R: 999.928 us
SS74_DATANODE_OP_WRITE_BLOCK_REQUEST
R: 12500.048 us
SS70_DATANODE_OP_WRITE_BLOCK_RECEIVE
R: 999.928 us
SS70_DATANODE_NEWBLOCKRECEIVER_START
R: 108000.040 us
SS70_DATANODE_NEWBLOCKRECEIVER_END
R: 999.928 us
SS70_DATANODE_OP_WRITE_BLOCK_REPLY
R: 12500.048 us
SS74_DATANODE_OP_WRITE_BLOCK_SUCCESS
R: 999.928 us
SS74_DATANODE_OP_WRITE_BLOCK_REPLY
R: 999.928 us
SS74_DFSCLIENT_OP_WRITE_BLOCK_SUCCESS
R: 6000.042 us
SS74_DFSCLIENT_SEND_PACKET
F: 18.dat
F: 0.dat
SS74_DATANODE_RECEIVE_PACKET
F: 1.dat
SS74_DATANODE_ACCEPT_PACKET
F: 2.dat
F: 3.dat
SS74_DATANODE_SEND_PACKET
F: 17.dat
F: 4.dat
SS70_DATANODE_RECEIVE_PACKET
F: 5.dat
SS70_DATANODE_ACCEPT_PACKET
F: 16.dat
F: 6.dat
SS70_DATANODE_SEND_ACK
F: 15.dat
F: 7.dat
SS74_DATANODE_RECEIVE_ACK
F: 8.dat
SS74_DATANODE_ACCEPT_ACK
F: 9.dat
F: 10.dat
SS74_DATANODE_SEND_ACK
F: 14.dat
F: 11.dat
SS74_DFSCLIENT_RECEIVE_ACK
F: 12.dat
SS74_DFSCLIENT_ACCEPT_ACK
F: 13.dat
R: 0.000 us
SS74_DFSCLIENT_END_BLOCK
(a)
(b)

Figure 6: This figure shows the before and after compression of a write graph. For parts of the graph that do not repeat, the edges are labeled with a single latency. Otherwise, if the transition occurs multiple times, the latencies are stored in a separate file and the edge is label with the file's name.

number of categories by combining them. The major problem is actually the fact that there are too few samples in each category for a valid statistical analysis. Therefore, if we can combine categories that are "similar", we increase the number of samples per category, allowing for a better statistical analysis. For example, if we look at HDFS writes again, we see that there are large number of potential categories because of all the permutations of the datanodes for the pipeline. However, it's likely all permutations of the same r datanodes produce very similar graphs. It may even be that any permutation of r datanodes in the same rack produce very similar graphs. Also, combining categories does not complicate diagnosis. If, after combining categories, we find a mutation in a combined category, we can perform anomaly detection in the category to find the individual category that contributed most to the mutation. Thus, it makes sense to combine graphs. Unfortunately we have not yet designed an algorithm to find combinable categories, and so we cannot yet evalute its effectiveness.

7 Implementation

This section describes the modifications made to the X-Trace instrumentation API, describes the modifications made to HDFS protocols in order to support X-Trace instrumentation and explains some of the HDFS behavior that has been instrumented.

7.1 X-Trace Modification

The base X-Trace instrumentation API was improved upon for this framework. One of the major changes is changing the method to uniquely identify reports, allowing for a robust method of stitching reports together to create request flow graphs. As mentioned before, the original method was to generate a new event Id for each new report. Then, one would add an Edge key to link back to the parent. The new method is slightly different and is based on a similar idea as implemented by Stardust [14]. New event IDs are only generated when necessary. Instead, we rely on timestamp to uniquely identify and order the reports. Given that timestamps are at the nanosecond time scale, it is not possible for two reports to be timestamped with the same value when on the same machine. There are still times in which the original method must be used. Since we do not assume that the clocks are synchronized in the cluster, we must change event IDs whenever a request moves to a different machine because the invariant is no longer true. Other cases include forks. Both branches of a fork must have a different event ID from the parent or one cannot capture the parallel activity. Regardless, now there exist cases where even some reports are lost, a portion of the graph can still be constructed.

Another important change is the addition of the job ID. Adding the job ID is necessary to identify HDFS request as being part of a MapReduce job and necessary to support the API described in the Design section. Finally, an important addition is request-level sampling [6], i.e. probabilistically choosing whether or not to report a request. By sampling requests instead of reporting all of them, we can keep the overhead of the framework low in both processing time and network traffic. In general, even though not all requests are reported, if we choose a good value for the sample rate, statistical performance diagnosis still can be done effectively as shown by Spectroscope when working with sampled request flow graphs generated by Stardust.

7.2 Protocol Modification

There are two main communication protocols in HDFS. On is Hadoop RPC, which is mainly used by client applications and datanodes to communicate with the namenodei in HDFS. Then there is the DataTransferProtocol, the protocol used for operations related to data blocks such as read, writes, and checksums.

7.2.1 Hadoop RPC

Hadoop has a RPC library used by all components that use RPC to communicate. This library uses Java's reflection capabilities to implement an RPC system. First an interface containing all the functions to be exposed by the server is written. The server implements this interface while the client creates a proxy object that acts on behalf of the interface. Whenever a function of the interface is invoked by the client, the proxy object captures the call and issues the RPC. The server receives the RPC and executes the function. The server sends the results to the proxy, and the proxy returns the results to the client. To add X-Trace support to this path, the X-Trace metadata needed to be added to each issued RPC and to each RPC reply. This was done essentially by concatenating the length of the metadata and the metadata itself to the end of the packet (similarly with job ID). Instrumentation points are generated automatically for any function using the RPC library. The only requirement is that an interface must declare to X-Trace that its functions should be instrumented.

7.2.2 DataTransferProtocol

DataTransferProtocol is the protocol used by HDFS to initiate block operations. To start a block operation, one sends a message that starts with the operation type (e.g. **`OP_READ_BLOCK`**) followed by any parameters required by the operation. Also part of the protocol is the specification for the packet used for block transfer. It has a fixed size header that is then followed by chunks of the block and their checksums. To add support for X-Trace instrumentation, the metadata (and job ID) were concatenated to the end the block operation messages, and the packet header was augmented to contain the metadata. Since the header was fixed size, the metadata size had to be a fixed size as well. In the current implementation, the task ID and event ID are both 8 bytes in length in all cases.

7.3 Instrumented Behavior

Currently, only requests made by client applications to HDFS are instrumented. The communication paths can be catagorized into two types: communication between clients and the namenode, and communication between clients and datanodes.

7.3.1 Client-Namenode

Communication of this type uses the RPC library. Because of the automatic instrumentation point generation, the only code required for these requests to be captured is the declaration as mentioned in Section 7.2.1.

7.3.2 Client-Datanode

Communication of this type uses the DataTransferProtocol. Only two types of block operations are instrumented at the moment, which are the block reads and writes as described in Section 3. This section describse the operations in greater detail.

Read Reads are initiated using **`open()`**. In **`open()`**, the client gets a list of the locations of the first few blocks from the namenode and caches them and then returns a **`DFSInputStream`** object. The client uses the **`read()`** method of the **`DFSInputStream`** to read the file. Communication with the datanode does not happen until a call to **`read()`** is made. When **`read()`** is called and it is not in the middle of reading a block, it calls **`blockSeekTo`**, which looks in the block location cache to find the location of the best datanode to read from for the next block. In the case of a cache miss, the client first communicates with the namenode to get the next few block locations. It then initiates a connection using the **`OP_READ_BLOCK`** command in the **`DataTransferProtocol`**. If the command is successful, a socket is set up such that the client can read from the block. Subsequent **read()**s read from the socket until the end of the block and either the process repeats itself or end of file was reached. Instrumentation points were added to the start and end of the above mentioned functions to capture the behavior. Figure 7 shows the start of a new block read.

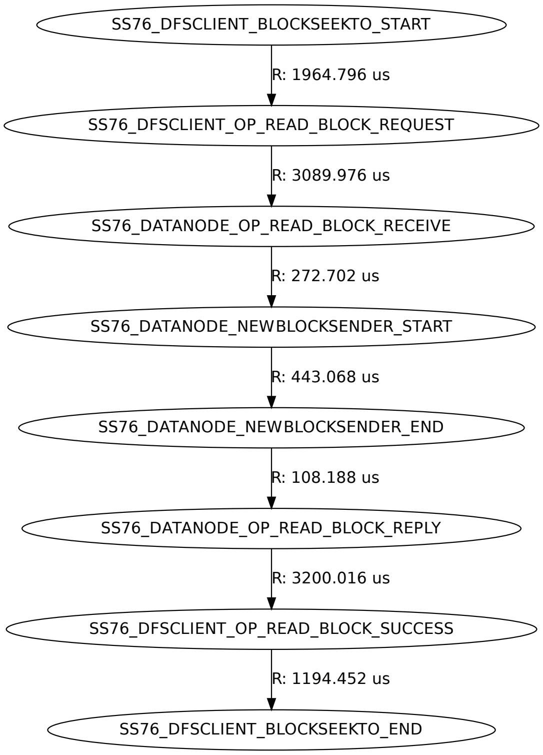

SS76_DFSCLIENT_BLOCKSEEKTO_START
R: 1964.796 us
SS76_DFSCLIENT_OP_READ_BLOCK_REQUEST
R: 3089.976 us
SS76_DATANODE_OP_READ_BLOCK_RECEIVE
R: 272.702 us
SS76_DATANODE_NEWBLOCKSENDER_START
R: 443.068 us
SS76_DATANODE_NEWBLOCKSENDER_END
R: 108.188 us
SS76_DATANODE_OP_READ_BLOCK_REPLY
R: 3200.016 us
SS76_DFSCLIENT_OP_READ_BLOCK_SUCCESS
R: 1194.452 us
SS76_DFSCLIENT_BLOCKSEEKTO_END

Figure 7: Starting a block read: The client sends a **OP_READ_BLOCK** message to the datanode. The datanode creates a new block sender, which is the object that sends the block to the client.

Write Writes are initiated using **create()** or **append()**. In both functions, **DataStreamer** and **ResponseProcessor** threads are created and then **DFSOutputStream** object is returned. The client uses the **write()** method of the **DFSOutputStream** to write to the file. When **write()** is called, it writes data to a packet to be sent by the **DataStreamer**. When the packet is full, it is placed on the **DataStreamer**'s send queue to be sent asynchronously. For each block, the **DataStreamer** makes a call to the namenode to get a set of datanodes to replicate the block on. It then initiates a connection with the first node in the pipeline with the **OP_WRITE_BLOCK** command. Then, it sends packets to the first datanode in the pipeline and places those packets on to the **ResponseProcessor** acknowledgement queue. For each datanode, if there is a next datanode, it first sends the packets to the next datanode and then writes the packet to disk and puts the packets on its own acknowledgement queue to wait for acknowledgments downstream. The last datanode sends acknowledgements for each packet it receives upstream. Acknowledgements eventually propagated back to the **`ResponseProcessor`** after each datanode verifies each acknowledgement along the way.

As mention in Section 4, block writes are the largest cause of both the large graphs and large number of categories problems. This is because of the large number of HDFS packets which have to be sent to complete a block write, since a block is 64MB and each packet is 64KB. This means at least 1024 packets need to be sent to complete the write. On the send of a packet to a datanode, 5 instrumentation points are touched. Hence, with a replication factor of 3, 15 reports are generated per packet leading to over 15,000 nodes in the graph for a single block write. Replication allows for $\Theta(n^r)$ paths for satisfying a block write request. Given that the namenode attempts to load balance data blocks across the datanodes, it is definitely possible every ordering of every set of nodes to be chosen for the pipeline causing a large number of categories to be formed.

8 Evaluation

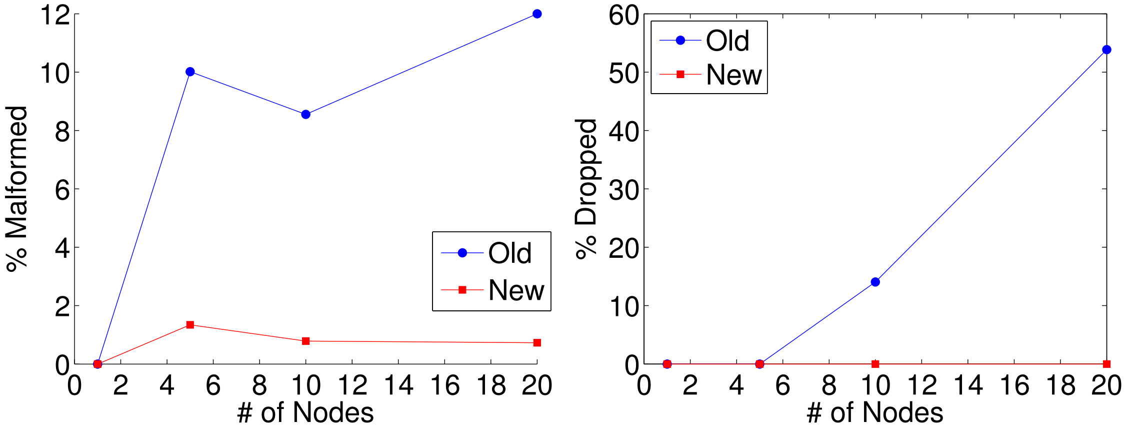
This section discusses the perfomance of the system, and compares the scalability of the original X-Trace collection backend compared to the new one that uses Flume with HBase. We show the average graph sizes and average number of categories for several MapReduce workloads, before and after graph compression. We also show that the instrumentation is correct and potentially useful by showing a small example of problem diagnosis using Spectroscope. All experiments, including the ones mentioned in Section 5, were run on a cluster of 20 machines each with 2 Intel Xeon 3.00GHz processors, 2GB of RAM, and 1Gbit Ethernet. On these machines, the UDP receive buffer was increased to 393,213 bytes, which is triple the original buffer size, in order to counter the effects of socket buffer overflow. The replication factor of the HDFS instance was 3. For these experiments, the X-Trace server's bounded buffer had a size 64 times the original, which allowed it to handle loads much larger than it normally would have. Both architectures used the modified stitching method mentioned in section 7.

8.1 Performance

Using a 100 GB sort workload over 10 nodes, we tested the overhead of instrumentation. The comparison was made been a cluster with no instrumentation and a cluster that samples requests at a rate of 10% (which is also the sample rate used for all experiments). It was found that, on average, the instrumentation induced an overhead of 2.5% with respect to job completion time, which matches the low overhead numbers of other end-to-end tracing implementations[13, 2, 14].

8.2 Scalability

We performed the same scalability experiments as described in Section 5 on the new architecture as we did on the original. The below graphs shows the values for both the original and new architectures. One can see that the new architecture behaves better than the original. In terms of malformed graphs, the number is actually constant across all cluster sizes, which is why the



(a) Comparing % of malformed graphs

(b) Comparing % of dropped records

Figure 8: Scalability graphs of both architectures

percentage decreases overall. There are no reported packet drops at all in the new system. The malformed graphs are primarily due to incomplete graphs collected at experiement end. From these results we can conclude that there was major benefit from switching to the new architecture.

8.3 Graphs

As noted in Section 4, there exist problems in both the graph size and number of categories. Tables 2 and 3 show the values we see from a 10-node cluster running several benchmark workloads on 100GB of input data.

Workload	Avg. Size	Avg. Write Size	Standard Deviation
sort	1625.153	15443.641	4740.376
wordcount	98.502	13309.571	1186.343
grep	7.354	41	1.408

Table 2: The average sizes of graphs generated by the workloads. Size refers to the number of nodes in the graphs

Workload	# of categories	Avg. Category Size
sort	289	5.249
wordcount	125	8.168
grep	117	8.932

Table 3: The average size number and size (i.e. number of graphs) of graph categories

8.4 Reduction

As we can see from the previous tables, the average graph size can be very large. Also, the number of categories can be large with few samples in each category. Using the algorithm described in Section 6, we were able to compress the graphs and achieve the values as shown in Tables 4 and 5.

Workload	Avg. Graph Size	Avg. Write Size	Standard Deviation
sort	10.769	41	10.389
wordcount	7.532	41	2.940
grep	7.354	41	1.408

Table 4: The average sizes of graphs after compression

Workload	# of categories	Avg. Category Size
sort	275	5.516
wordcount	124	8.234
grep	117	8.932

Table 5: The average number and size of graph categories after compression

By compressing the graphs, we have significantly lowered the average graph size of writes, essentially solving the problem of large graphs, though diagnosis tools may need to make modifications in order to be compatible with the changed format. However, we see that the number of categories was barely reduced. We believe that our approach to graph category compression will exhibit similar benefits for graph category number and size, but do not yet have an implementation with which we can experiment.

8.5 Problem Diagnosis

In order to show that Spectroscope can be used, we ran two executions of the same 100GB Sort workload on 10 nodes. One execution was a normal execution, which represented the non-problem period. The other execution had an induced delay in the RPC library and represented a problem period. The experiment was done with both a 50ms delay and a 500ms delay. For both values of delay, Spectroscope was able to detect response-time mutations in the affected categories and saw no mutations in any of the others. The p-value calculated by Spectroscope was less than 0.001 for both delay amounts for each of the categories. This means that Spectroscope was confident in rejecting the null hypothesis for those categories.

We also modified Spectroscope to be able to accept compressed graphs. Using compressed graphs, Spectroscopes total processing time was reduced from several hours to several minutes. We also discovered an unexpected result. We ran a similar Sort experiment where instead, we induced a delay in a node's HDFS packet sends so that the latencies in write graphs would be affected. Spectroscope was unable to detect this delay using uncompressed write graphs because it believed that there were too few samples for each edge to detect response-time mutations. However, when the graphs were compressed, Spectroscope detected this behavior change with very high confidence (p-value < 0.0000001). This seems to be the result of labeling edges with a set of latencies. Spectroscope considered each of the latencies a separate sample. As a result, there were over 1000 samples for many of the edges, which is significantly greater than what is necesary for a meaningful hypothesis test and why Spectroscope had such high confidence. Thus, it would seem that graph compression can aid in problem diagnosis by reducing graph size as well as increasing the number of samples for sparse categories.

9 Future Work

There are several main areas in which we see valuable additional work. This section lists and explains the areas which we wish to expand into. Most of the future work will be focused on improving the framework in various ways. However, there will also be work looking into solving the graph problems encountered during development.

9.1 Instrumentation

Further instrumentation should be added HDFS to capture the namenode-datanode and some other parts of the datanode-only communication path. HDFS does perform system tasks in the background, which could affect performance of jobs. Another place for instrumentation are locks in HDFS, since locks are places of contention and are often the causes of performance problems.

9.2 Case Studies

Case studies should be done to validate the usefulness of the end-to-end tracing framework. There are plans to introduce the framework into a production cluster. Feedback from users of the cluster should be able help improve the effectiveness of the framework. Also, we would like to test the capabilities of the framework to capture previously seen performance problems in HDFS and/or Hadoop and be used to determine the root cause.

9.3 Using MapReduce to convert traces into graphs

There are plans to add a periodically running MapReduce job to the framework. What this MapReduce job would do is read from the record table and automatically construct the request flow graphs and insert them into a separate table as rows. Then, the query API would be augmented to allow users to query for complete graphs instead of just activity records. There are several challenges here. One is a question about knowing when all the reports for some request have been received. When the reducer constructs the graph and realizes that it is malformed, it does not know if it is malformed because not all reports have been received yet or because some report was dropped. And, in some instances, it is possible to construct a non-malformed graph even when not all the reports have been received. A simple solution is to run the job on all the rows in the record table every time and then update the graph table when changes are found. However, this will lead to poor performance as the record table will continually increase in size. It makes sense to only require construction of graphs of new requests in order to improve performance. Another challenge is deciding on a practical schema to store the graphs. It is not entirely clear how one would efficiently store a graph as a single row in a table. As we have seen, graphs can get very large and thus a single row may have to store a lot of information. At the moment, we have map and reduce functions that can take activity records and convert them into in-memory representations of graphs. Most of the challenges still require futher work.

9.4 Graph Challenges

The graph compression algorithm used to gather the results in the evaluation section attempts to reduce request flow graphs into a state-machine representation to reduce graph size. It is not clear that this is the best method for compressing graphs. There may exist other methods that may work better. One possibility is to predefine an expected final graph structure instead of trying create one automatically, like the state machine method, and attempt to make the large graph fit into the graph structure using some heuristics. Such a method might allow for better compression, but it requires a deep understanding of the instrumented system. Therefore, we must weigh the benefits and drawbacks of each method to determine the best method for compression.

At present, we have not yet designed an algorithm for finding similar catagories and combining them. In general, such an algorithm will rank the similarity of two categories by some measure. It is not clear what the correct measure is. As mentioned before, the change in variance by combining categories might be a good measure. However, there might be other measures which are better or could be used in tandem to improve the algorithm. More research is necessary to develop the algorithm.

10 Conclusion

In this report, we introduce the design and implementation of an end-to-end tracing framework and its integration into HDFS. Our approach to implementation differs from previous approaches mainly in the desire to have detailed low-level instrumentation. We have noted several challenges that were encountered during development. Some of these challenges involve the request flow graphs themselves and their sizes and categories. These challenges may appear in other systems and therefore are important to continue researching. As our evaluation has shown, the current implementation uses Flume with HBase and scales well up to 20 nodes, losing essentially no records. We have also shown promising results on the compression of large graphs and a case where the instrumentation could be used to diagnose a simple performance problem. There are still many things that can be improved upon, but, the preliminary evaluations show there is potential for this end-to-end tracing infrastructure to be effective and useful.

References

[1] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa minor: versatile cluster-based storage. In *Conference on File and Storage Technologies*, pages 59–72. USENIX Association, 2005.

[2] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Symposium on Operating Systems Design and Implementation*, pages 259–272. USENIX Association, 2004.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Symposium on Operating Systems Design and Implementation*, pages 205–218. USENIX Association, 2006.

[4] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *Symposium on Networked Systems Design and Implementation*, pages 309–322. USENIX Association, 2004.

[5] Flume. http://www.cloudera.com/blog/category/flume/.

[6] Rodrigo Fonseca, Michael J. Freedman, and George Porter. Experiences with tracing causality in networked services. In *Internet Network Management Workshop / Workshop on Research on Enterprise Networking*. USENIX Association, 2010.

[7] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: a pervasive network tracing framework. In *Symposium on Networked Systems Design and Implementation*. USENIX Association, 2007.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM Symposium on Operating System Principles*, pages 29–43. ACM, 2003.

[9] Graphviz. http://www.graphviz.org/.

[10] Hadoop. http://hadoop.apache.org/.

[11] Hbase. http://hbase.apache.org/.

[12] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *Symposium on Networked Systems Design and Implementation*, pages 43–56. USENIX Association, 2011.

[13] Benjamin H. Sigelman, Luiz Andre 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, 2010.

[14] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. Stardust: Tracking activity in a distributed storage system. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 3–14. ACM, 2006.

[15] Matei Zaharia and Andy Konwinski. Monitoring hadoop through tracing, 2008. http://radlab.cs.berkeley.edu/wiki/Projects/X-Trace_on_Hadoop.