MTracer: A Trace-oriented Monitoring Framework for Medium-scale Distributed Systems

Jingwen Zhou*†, Zhenbang Chen*†, Haibo Mi*, and Ji Wang*†
*Science & Technology on Parallel & Distributed Processing Lab, National Univ. of Defense Technology, Changsha, China
†College of Computer, National Univ. of Defense Technology, Changsha, China
Email: zjw8612@gmail.com, zbchen@nudt.edu.cn

***Abstract*—Trace-oriented runtime monitoring is a very effective method to improve the reliability of distributed systems. However, for medium-scale distributed systems, existing trace-oriented monitoring frameworks are either not powerful or efficient enough, or too complex and expensive to deploy and maintain. In this paper, we present MTracer, which is a lightweight trace-oriented monitoring system for medium-scale distributed systems. We have proposed and implemented several optimizations to improve the efficiency of the monitoring server in MTracer. A web-based visualization is also provided in MTrace to visualize a monitored system from different perspectives. We have validated MTrace in a real medium-scale environment. The experimental results indicate that MTrace has a very lower overhead, and can handle more than 4000 events per second.**

***Keywords*—monitoring; trace-oriented; visualization; parallelization; distributed system**

I. Introduction

With the improvement of computing power and the reduction of price, many corporations and institutions build their own clusters that are medium-scale and contain tens to hundreds nodes. Various distributed systems are deployed on these clusters, bringing enormous convenience and benefits.

However, due to the problems resulted from software and hardware, such as hardware errors, software bugs and network issues, failures often happen in distributed systems, and cause functional or performance problems. What is even worse is that some of these failures are very hard to locate and replay. For example, on July 29, 2012, Amazon suffered a service disruption in one of its Availability Zones in the US East Region, which lasted about 20 hours and affected lots of companies and websites [1]. Therefore, how to ensure the reliability of these distributed systems is a really important problem. Because real world distributed systems are usually very complex, traditional methods, such as testing, verification and validation, cannot solve the reliability problem at the design stage. Some failures of a distributed system appear after deployment. Hence, runtime monitoring complements the methods used at design stage as a method for improving the reliability of distributed systems at runtime. By using runtime monitoring, we can monitor a distributed system to record the runtime information of the system, which can be used for many activities, including online bug detection, failure localization, runtime enforcement, *etc*.

Currently, there are two kinds of monitoring for distributed systems: resource-oriented and trace-oriented. Resource-oriented monitoring [2], [3] usually tracks the hardware resource consumption of a distributed system, such as the memory and CPU. In contrast, trace-oriented monitoring [4]–[11] tracks the execution paths, or called traces, of the requests in a distributed system. Trace records the context of each step in a request, like latency, and relationships between steps, such as function calls and process communications. Generally, we consider that trace-oriented monitoring collects more valuable information than resource-oriented monitoring if we want to understand a system in more detail. In addition, traces can also reflect resource consumptions to some extent.

Today, there already exist some available platforms for trace-oriented monitoring, such as X-Trace [4] and P-Tracer [5]. However, some platforms, such as X-Trace, are still in prototype, which are not efficient and powerful enough for medium-scale systems. On the other hand, some platforms are very complicated to deploy and maintain, especially when we only want to monitor a medium-scale distributed system. Sometimes it is even more complex than the monitored systems, *e.g.*, P-Tracer uses a map-reduce process to construct call trees [5]. Actually, in these monitoring platforms, failures would also easily occur and are also difficult to recover. Therefore, we believe what many medium-scale distributed systems need is a trace-oriented monitoring system that is *effective* and *efficient* for using, and *lightweight* for deployment and maintenance.

In this paper, we propose a monitoring framework, called MTracer, to help understanding the behaviors of medium-scale distributed systems and also the detection, locating and recovery of failures. MTracer has the following features. (1) **Lightweight**. MTracer adopts the server-client framework. The resource needed by running the monitoring server is little, and the overhead of a client is negligible. Thanks to the simple framework, exceptions seldom happen in MTracer, and it is easy to recover in case of failures. (2) **Efficient**. When receiving the events from clients, the server stores them in parallel. Optimizations are also introduced to improve the efficiency. Our experimental results show that MTracer can process 4000+ events per second, which is 7+ times than the original version. (3) **Real-time**. Clients send events asynchronously, and the server can generate trace trees quickly. Hence, users can inspect the behaviors of the system under monitoring in time. (4) **Visualized**. We also provide a friendly web-based interface for visualization. Users can do queries from different aspects, such as trace, operation and node. Other functions, such as trace classification and outlier highlighting, are also integrated.

To summarize, this paper makes two main contributions. First, we have built a trace-oriented monitoring framework

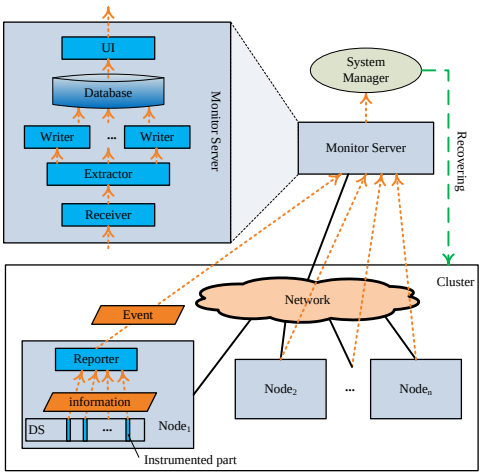
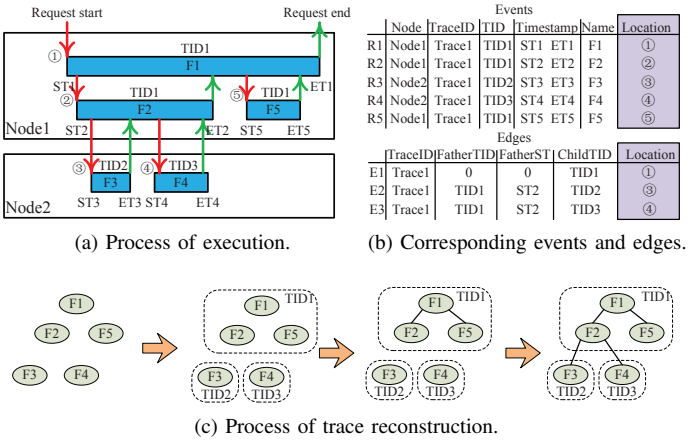


Fig. 1. The architecture of MTracer. The blue boxes represent the components belonging to MTracer, while the orange directed dashed lines represent information flows.



(a) Process of execution. (b) Corresponding events and edges.

(c) Process of trace reconstruction.

Fig. 2. An sample of trace recording and reconstruction.

that is efficient, lightweight and real-time, to collect the trace information of medium-scale distributed systems. Second, a corresponding interface for visualization is implemented to help users inspect the behavior of their systems and do failure locating and recovering.

The rest of this paper is structured as follows. In Section II, we give the architecture of our approach. Section III describes the trace recording and reconstruction. In Section IV, we discuss the process of data storing, focusing on the optimizations. Section V discusses some aspects of the visualization. In Section VI, we present our experimental results. Section VII reviews the related work and Section VIII concludes.

II. System Architecture

Usually, the main task of a distributed system is to handle user requests, and a trace is the internal execution path of handling a request. Hence, traces record the behaviors of a distributed system. Basically, a trace is composed by events, which record the context of each step in handling a request, such as function name and latency, and the relationships between events, like function calls. Thus, we take events and their relationships as the minimal units for data collecting and storing. Fig. 1 shows the architecture of MTracer, which is in the typical client-server style, and contains a monitoring server and many clients inside. DS is a distributed system being monitored, and deployed on n nodes. Each node acts as a MTracer client, collecting information and sending them to the monitoring server.

In more detail, we need first instrument the distributed system, using the interfaces provided by MTracer at the places where we are interested in to collect the information we want. When a request passes these places, the related information is gathered and packed into an event. The reporter sends the event to the server through network in terms of UDP or TCP protocol, decided by user. Once receiving an event, the monitor server extracts the useful information and delivers them to different writers. The writers then store the information concurrently. To speed up the process of data storing, we also introduce two optimizations. The web-based UI can construct the causal relationships between the events in a same request to form a trace that records the internal execution path of the request. With request traces, users, such as system managers, can inspect how each request proceeds. In addition, we can also use trace information to support online fault detection, localization and fixing.

In summary, MTracer contains three parts: data collecting, storing and visualizing, which will be explained in detail in Section III, Section IV and Section V, respectively.

III. Trace Recording and Reconstruction

As mentioned earlier, a trace consists of events and the relationships between the events. An event contains some information collected automatically, such as start/end time stamp and host address, at an instrumentation point. In this section, we focus on how to record the relationships, which are the most important for trace reconstruction. A procedure for reconstructing a trace is also introduced.

A. Trace Recording

To distinguish different traces, we assign a unique TraceID to each request. We also introduce TID for events. For overhead consideration, inspired by the idea of P-Tracer [5], we do not assign a new TID for each event. A TID can be understood as a temporary thread ID, and we can identify the events with a same TID with respect to the start time stamps. The rules of recording trace, especially event relationships, are as follows.

- Assigning a new TID when a trace starts.
- Each time a node communicates with a remote node, assigns a new TID for the remote node, and preserves the local TID.
- When generating an event, records local TID, together with the start and end time stamps.
- The first time of generating an event after a new TID assigned, an additional event also created, called edge[1], recording the information of the causal relationship.

[1]Actually, an edge is sent together with the corresponding event in one network packet.

Fig. 2(a) illustrates the execution process of an example request, involving two nodes. Blue boxes represent functions, and red arrows mean the startings of invocations, while green means finishing. ST and ET abbreviate for start time and end time, respectively. The numbers, also appeared in the column of Location of Fig. 2(b), indicate the locations of generating events and edges. Each one is explained as follows: (1) A new trace starts, hence MTracer assigns a new TID, which is TID1, to Node1, and a new event R1 generated subsequently, recording TID1, ST1 and other information. Since R1 is the first event after TID1 generated, an additional edge E1 is also created, recording the relationship between the current event and the former event, which is null for TID1. (2) Since this is not the first time of generating an event after TID1 has created, only R2 is generated. (3) F2 calls the remote function F3 on Node2, and a new TID, *i.e.*, TID2, is assigned to Node2, and both R3 and E2 are generated. (4) Another remote call induces another new TID, and R4 and E3 are generated. (5) Only generates R5. By the way, an event is usually sent when the invocation finishes.

B. Trace Reconstruction

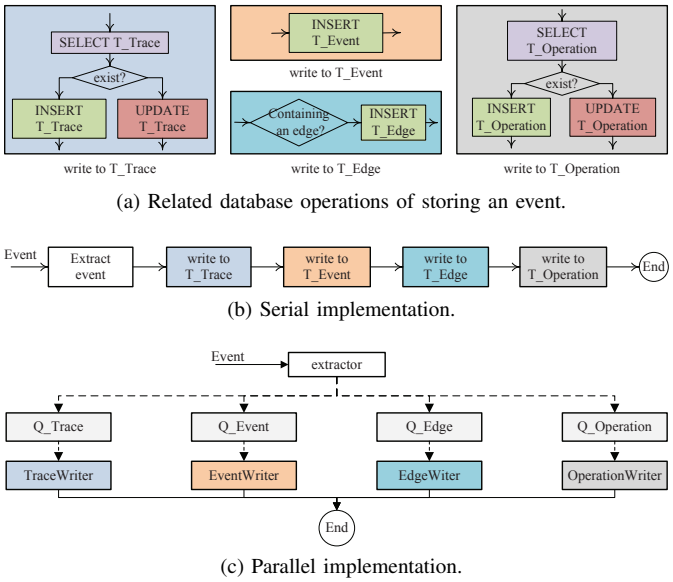
According to the events and edges received, we can reconstruct a trace on the server. The reconstruction procedure is demonstrated in Fig. 2(c). Each step is explained as follows:

- Pick out all the events and the edges with a same TraceID.
- Classify the picked events to different classes with TID, *i.e.*, all the events in one class have a same TID.
- Calculate the relationships of the events in each class according to the time stamps. Since all the events from one class are generated at a same node, the time stamps can be compared with each other, avoiding the famous problem of clock deviations in distributed systems. For example, (R1.startTime $<$ R2.startTime) and (R1.endTime $>$ R2.endTime) mean the operation F1 starts earlier and finish later than F2, thus F1 is the ancestor of F2 with respect to method call relation. Because there is no other operation that is also the ancestor of F2, F1 is the father of F2.
- Construct the relationships between classes using edges. The event, identified by the fatherTID and fatherST fields, is the father of all the events in the class decided by the childTID field. For example, F2 is the father of F3 and F4.

Therefore, a trace can be expressed as a tree. A node in a trace tree represents the execution of some code segments, which are usually functions. The edges in the tree have many meanings, such as F1 calls F2, F1 contains F5, and F2 communicates with F3. In general, we say F1 calls F2 if F1 and F2 belong to a same thread, and F1 triggers F2 if not.

IV. Data Storing

Because we are using the client-server architecture, the performance of our monitoring framework is basically determined by the performance of the monitor server. When receiving an event, the server first extracts the information, and then stores the information. Currently, we use database for storing traces.



(a) Related database operations of storing an event.

(b) Serial implementation.

(c) Parallel implementation.

Fig. 3. The implementation of data storing.

In our current implementation, there are following four tables for trace recording: T_Trace, which records the summaries of traces, such as title and the number of events/edges; T_Event, which records the details of each event, including name, start/end time, description, *etc*; T_Edge, which stores the context of edges; T_Operation, which contains the summary of each operation, *e.g.*, the max/min/average latency of the operation.

After extracting the information from an event, we will execute some data storing operations, which are related to above four tables, *e.g.*, a new event will always cause the insertion of a new record into T_Event and an update in T_Operation. All the related operations for recording an event are illustrated in Fig. 3(a).

A. Parallelization

A simple idea of recording an event is to carry out the operations in Fig. 3(a) in sequence, as shown in Fig. 3(b). However, this approach is really an *inefficient* method, since at least five database operations (2 SELECTs, 3 INSERTs or 2 UPDATEs and 1 INSERT) are required in just one event processing. In addition, this sequential method cannot be easily parallelized, because a table is locked when being updated.

Based on the fact that the operations on different tables can be carried out concurrently, we propose a concurrent method for storing events, which is shown in Fig. 3(c). In principle, we divide the operations according to tables. We maintain a queue and a writer for each table. When an event arrives, the extractor extracts the information and distributes the necessary update information to different queues, then the writer of each queue fetches the information from the queue and writes to the corresponding table. Based on this idea, the writers and the extractor can work in parallel, which can improve the performance of the monitoring server.

B. Optimizations

Actually, database operations dominate the execution time of the monitoring server, usually more than 90%. And, we observe that it is not necessary to store an event immediately after received. Hence, in addition to the preceding concurrent method, we also propose two optimizations to speed up the storing procedure, whose basic idea is to reduce the times of database operations.

1) Batch inserting: The operations of T_Event and T_Edge are only INSERT operations. Hence, we can use the batch inserting of database to insert events and edges in a batch style. Batch inserting inserts many records to a table in one time, which is much more efficient than one by one. Take T_Edge for example. Instead of inserting the edges from Q_Edge one at a time, the writer, *i.e*, EdgeWriter, flushes all records in Q_Edge when the time from last flushing exceeds t_{edge} or the number of queued edges reaches n_{edge}, using batch inserting.

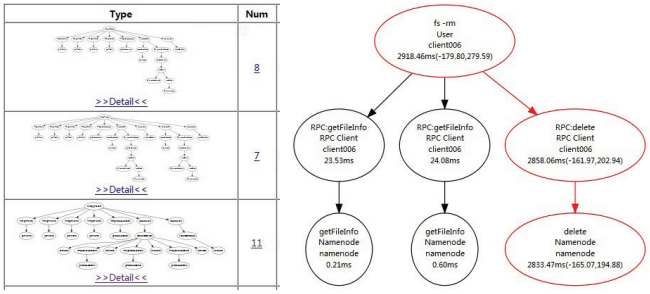
The values of n_{edge} and n_{event} should be chosen properly. Too small, the benefit of batch inserting is not significant; too big, the records in the queue would increase very fast during batch inserting, resulting in losing data. In our experiences, we set both of them to 10. With the same reason, choosing the values of t_{edge} and t_{event} should consider the balance between effectiveness and realtime requirements, and we set both to 1 second.

2) Information updating in memory: Furthermore, based on the parallelization method, we can optimize the procedure of updating trace and operations information. Because some information will be kept in the queue for a while before updated to database, we can do the updating directly in memory when it is still in the queue. For example, when receiving an event, which needs to update the information of the corresponding trace, such as increasing the event number field by 1, if the trace is still in Q_Trace, we can do the updating directly in the memory, saving one time of database querying and updating. As in previous optimization, when time out or queued elements number reaches the threshold, we flush all records in the queue.

The effectiveness of this optimization depends on the situations of receiving events. Take T_Trace for example. If many received events belong to few traces, this optimization will save many database operations. Oppositely, if all the events are from different traces, there would be no effect. Actually, in practice, we get the former situation in the actual applications, *i.e.*, the events received in a short period likely belong to a same trace. By the way, n_{trace}, $n_{operation}$, t_{trace} and $t_{operation}$ should also be chosen properly.

V. Visualization

Based on the recorded traces of a monitored system, we can inspect the system from different perspectives. Basically, the inspections can be implemented as different queries to the database. We have provided a web-based visualization frontend for visualizing different queries, such as the operations executed in a certain node and the traces generated during last hour. In addition, some kinds of deeper data analyses, such as the distribution of the operations in each node and abnormal traces, are also supported. Due to the page limit, we only introduce three of them.



(a) Trace classification. (b) A performance exception sample.

Fig. 4. Samples of visualization.

A. Trace Tree

Trace tree is a nature form for visualizing traces. A trace tree clearly shows the details of the execution, including the function calls, the communications between nodes, the latency of each operation, *etc*. Fig. 4(b) displays a trace tree for a request that is a removing request of a file in HDFS [13], which we use as a case that will be described in the next section. According to the trace, we can observe that the client gets the information of the file from the Namenode first and then asks for removing it. Three remote procedure calls (RPCs) are made by client006, and the latency of each operation is also marked in the tree.

B. Trace Classification

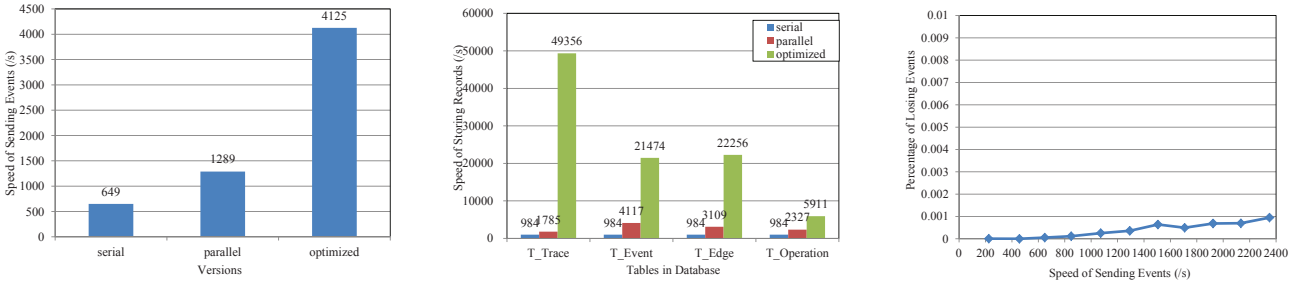
Since the requests of a same type may generate different traces, we can classify the traces of a same request kind according to the topological structures of trace trees. Based on the topological structures, we can understand different behaviors, or even functional exceptions, of a same request type. In Fig. 4(a), the first two topologies represent the two cases of the file read request in HDFS with different file sizes, and the third one shows a failure. The reason of the failure is that all the datanode processes are killed, and the client can only get the file information from Namenode, but cannot connect to Datanodes to get any data block. Hence, the request failed. In addition, the number and the percentage of the traces with a topology are also given, hence we can know the frequency of a topology that the traces of a request type will be.

C. Performance Problem Diagnosis

A performance problem diagnosis algorithm [12] is implemented to locate the root causes of performance problems. With the latency information recorded, we employ a PCA-based analysis [12] to list the outliers of the traces with a same topology, and mark the likely exception operations. Fig. 4(b) shows an example of a trace tree with a performance problem, and the red path is the suggested root cause. Note that, the normal latency intervals are also shown next to the latency of an exception node, to indicate how abnormal the latency of the operation is.

VI. Implementation and Experiments

MTracer has been implemented in Java using MySQL [15] as the database. The reporter, the receiver and the event



(a) Maximal speeds of different versions. (b) Maximal speeds of different tables. (c) Percentage of losing events on network.

Fig. 5. Experimental results.

translation protocol are adopted from X-Trace [4]. Other parts, such as data collecting, storing and visualizing are totally rewritten. The web-based UI is implemented in JSP and use Graphviz [16] for drawing trace trees. Although we deploy MTracer only on Linux, we believe it also works on other OSs due to the independence of Java.

We have carried out extensive experiments to validate MTracer in three aspects: *overhead*, *effectiveness* and *usability*. We carried out our experiments in a real environment, which is composed by the virtual machines (VMs) hosted on our private IaaS platform. The virtual machines can be divided into three types: small instance, with 0.5GHz CPU and 0.5GB memory; medium instance, with 4x1GHz CPU and 1GB memory; large instance, with 8x1GHz CPU and 2GB memory. The lengths of the queues in MTracer are all set to 1024. The time-out and queued record thresholds of each queue are set to 1 second and 10, respectively.

A. Overhead

Basically, the overhead of the clients on the nodes of a monitored system is our main concern. Since the overhead to the monitored system is related to many factors, such as the instrumentation number and the collected information, we evaluate the overhead of generating one event, instead of the influence to the whole system. We generated 10,000 events for collecting elementary information, such as node address and time stamps, and then take the average values as the results. The experiments are carried out on small instance VMs.

The average time of generating an event is 0.046ms, which is pretty acceptable, since many operations in distributed systems need seconds or minutes to finish. The size of an event is 0.315KB, which consumes less than 2MB bandwidth even the monitor server reaches the upper limit, and 2MB is negligible comparing to the GB-level network in the environment. It is necessary to point out that generating random IDs is time-consumed, which needs 0.057ms for each, and is even more than generating an event. Fortunately, the strategy in MTracer avoids generating a new ID for each event, which can reduces the generations of lots of IDs, at least 50% less in our case. Actually, more local invocations in a trace induce greater reduction of IDs.

B. Effectiveness

Several methods are introduced to speed up the monitor server. In this subsection, we discuss the benefits brought by these methods. We use a large instance VM for the monitor server and medium instance VMs for clients. The clients are launched at the same time, and send events to the monitor server concurrently. The speeds of sending are the same on all clients, about 214 events per second, so that the speed of receiving events on the monitor server can be calculated with the number of clients. To decide whether the server reaches its upper limit, we monitored the inner data structures. Once the server starts to lose events, we consider that it reaches the maximal speed.

Fig. 5(a) displays the comparison of the maximal speeds of different MTracer versions, where serial means the version updating database in sequence, parallel represents the version implemented in parallel but without optimizations, and the optimized means the parallel version with optimizations. The parallel version doubles the speed of the serial version, and the optimized version is about 7 times better. In other words, the optimized version can process more than 4000 events per second, which is basically enough for a medium-scale system. In addition, proper values of the parameters in optimizations, decided by the monitored system and the environment, may bring a greater improvement in performance.

Fig. 5(b) shows the maximal speed comparison of each version for different tables, which demonstrate the effectiveness of the optimizations, *i.e.*, storing more information to database at one time. The effect on T_Trace is the best. The performance of optimization is 50 times better than that of the serial version, which validates our intuition, *i.e.*, the events received in a short period are likely belong to one trace. Batch inserting brings a same improvement to T_Event and T_Edge, 22 times, which may be better with a more proper parameter. The least improvement is at T_Operation, only 6 times, which limits the whole efficiency. This is because the events received in a short period do not likely contain a same operation.

In addition, we used UDP protocol to send events in our experiments, where the percentage of losing events on network is pretty low, less than 0.1%. Of course, the TCP pattern can be selected if we do not want to lose events.

C. Usability

For validating the usability of MTracer, we instrumented HDFS in Hadoop [13] to monitor the RPCs between the clients and the Namenode, and the data accessing processes between clients and the Datanodes. The experiment environment contains 50 Hadoop clients deployed on medium instance VMs,

keeping sending different HDFS requests in various speeds, 50 Datanodes together with a Namenode deployed on medium instance VMs too, and a monitoring server deployed on a large instance VM. We also injected 14 faults into HDFS, including functional and performance faults, such as data block missing, network slowdown and datanode suspending. MTracer can easily handle the generated events, and the UI correctly visualizes the process of each request. The functional exceptions caused by functional faults can be detected by classifying traces, and the diagnosis algorithm can locate a part of performance exceptions. Based on the theoretic calculation, we believe that, with such an instrumentation, MTracer can handle a HDFS cluster containing more than 200 nodes, with a real workload reported by Ali-Hadoop [14].

VII. Related Work

The most related work is X-Trace [4]. Actually, our implementation is based on X-Trace. X-Trace captures the causal path of a request crossing several network protocols and then results a corresponding trace tree. Compared with X-Trace, MTracer is more efficient because of the trace recording method and the optimizations. In addition, X-Trace stores each collected trace in a text file, hence suffers inflexible querying, difficult management and safety problems. Finally, MTracer provides a better visualization.

P-Tracer [5] is a performance profiling platform for large-scale cloud computing systems, which records the request traces of systems online. P-Tracer provides a suite of web-based UI to query the statistical information of cloud services for understanding the underlying cloud systems. P-Tracer generates call trees using a map-reduce process, and stores trace information in a key-value store system, which makes it sometimes not easy to be adopted for medium-scale systems. Same problem also exists for Zipkin [6], which is also a distributed tracing system released by Twitter.

AppInsight [10] records the asynchronous and multi-threaded nature for mobile apps. Because only system calls are concerned with, AppInsight instruments apps in a fully automatic manner, without requiring the source code or modifying the OS. We believe MTracer can also be used as the monitoring platform in the context of AppInsight. In addition, AppInsight also provides a useful hint for our future case studies.

In addition, there also exist some black box monitoring platforms, such as Project5 [11], that do not require the source code of a monitored system, and use reasoning methods to generating request traces. In practice, these approaches are less flexible and less precise than the ones based on white box.

VIII. Conclusion and Future Work

We present a trace-oriented monitoring system, MTracer, for medium-scale distributed systems. MTracer is lightweight and efficient. A frontend for visualizing request traces is also provided to help system understanding and inspection. Through the application on a real world distributed system in a real environment, we validate the efficiency and the usability of MTracer.

For future work, we will explore an easier way for instrumenting. In addition, since HDFS is used as the application system, maybe we could collect a set of user request trace datasets, which can be used as the datasets for the request trace-based research.

Acknowledgment

This work is fully supported by the National 973 Program of China under the Grant No.2011CB302603.

References

[1] Amazon, "Summary of the AWS service event in the US east region," http://aws.amazon.com/message/67457/, 2013.

[2] M. L. Massie, B. N. Chun., and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," Parallel Computing, vol. 30, no. 7, pp. 817–840, 2004.

[3] J. Boulon, A. Konwinski, R. Qi, A. Rabkin, E. Yang, and M. Yang, "Chukwa, a large-scale monitoring system," in Proc. of CCA, 2008.

[4] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-Trace: A pervasive network tracing framework," in Proc. of USENIX NSDI, 2007, pp. 271–284.

[5] H. Mi, H. Wang, H. Cai, Y. Zhou, M. R. Lyu, and Z. Chen, "P-Tracer: path-based performance profiling in cloud computing systems," in Proc. of IEEE COMPSAC, 2012, pp. 509–514.

[6] Twitter, "Zipkin," http://twitter.github.com/zipkin, 2013

[7] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, *et al*., "Dapper, a large-scale distributed systems tracing infrastructure," Tech. Rep., Google, 2010.

[8] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for request extraction and workload modelling," in Proc. of USENIX OSDI, 2004, pp. 18–33.

[9] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, *et al*., "Stardust: tracking activity in a distributed storage system," ACM SIGMETRICS Performance Evaluation Review, vol, 34, no. 1, pp. 3–14, 2006.

[10] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. "AppInsight: mobile app performance monitoring in the wild," in Proc. of USENIX OSDI, 2012, pp. 107–120.

[11] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," ACM SIGOPS Operating Systems Review, vol. 37, no. 5, pp. 74–89, 2003.

[12] A. Lakhina, M. Crovella, and C. Diot, "Diagnosing network-wide traffic anomalies," ACM SIGCOMM Computer Communication Review, vol. 34, no. 4, pp. 219–230, 2004.

[13] Apache, "Apache Hadoop," http://hadoop.apache.org/, 2013.

[14] L. Liang, "Ali Hadoop cluster architecture and service system," Conference report, Hadoop & Bigdata Technology Conference, 2012.

[15] MySQL, "MySQL: the world's most popular open source database," http://www.mysql.com/, 2013.

[16] Graphviz, "Graphviz: Graph visualization software," http://www.graphviz.org/, 2013.