Journal of Computer Science Research
http://ojs.bilpublishing.com/index.php/jcsr

ARTICLE

AgamottoEye: Recovering Request Flow for Cloud Systems via Log Analysis

Jie Lu[1] Feng Li[2] Lian Li[1*]

1. SKL of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, University of Chinese Academy of Sciences, China
2. Institute of Information Engineering, Chinese Academy of Sciences, China

ARTICLE INFO

Article history
Received: 23 May 2019
Accepted: 29 June 2019
Published Online: 30 July 2019

Keywords:
Cloud applications
Log analysis
Request flow

ABSTRACT

Cloud applications are implemented on top of different distributed systems to provide online service. A service request is decomposed into multiple sub-tasks, which are dispatched to different distributed systems components. For cloud providers, monitoring the execution of a service request is crucial to promptly find problems that may compromise cloud availability. In this paper, we present AgamottoEye, to automatically construct request flow from existing logs. AgamottoEye addresses the challenges of analyzing interleaved log instances, and can successfully extract request flow spread across multiple distributed systems. Our experiments with Hadoop2/YARN show that AgamottoEye can analyze 25,050 log instances in 57.4s, and the extracted request flow information is helpful with error detection and diagnosis.

1. Introduction

As we now enter into the cloud era, more and more applications are moving from local to cloud settings. These modern cloud applications are implemented on top of various distributed systems. A service request is decomposed into multiple sub-tasks, which are then dispatched to different components of various distributed systems. The different components across multiple systems interact with each other to render a service.

Request flow [10] depicts the detailed work-flow in processing a user request, which consists of causally-related activities across multiple components of distributed systems. Precise request flow information is useful for many important use cases, including anomaly detection [5], performance tuning [11], and system understanding [3]. For cloud providers, monitoring the request flow of a service request is crucial to promptly find problems that may compromise cloud availability.

We develop AgamottoEye, a new tool to automatically recover request flow from existing logs. Compared to those approaches which instrument and trace cloud applications to construct request flow [9], the log-based approach is non-intrusive, and can be easily adopted. Our one-year long study of manually tracing different distributed systems using Xtrace [6] also shows that there frequently exists log points around the manually instrumented trace point.

To automatically construct request flow from existing logs, AgamottoEye addresses the following challenges:

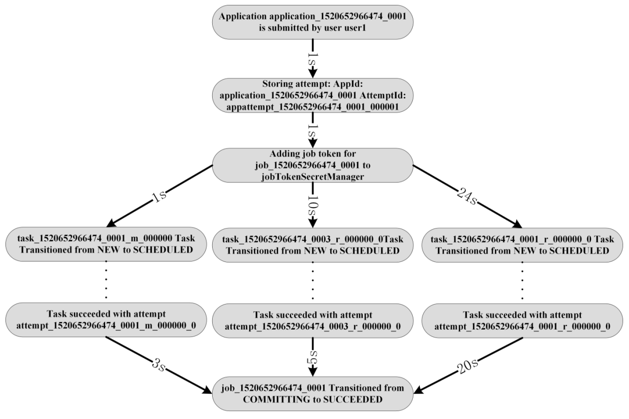
(1) Interleaved log instances. Cloud applications can serve thousands of user requests in parallel, and log messages from different requests are interleaved in log files. Furthermore, a

**Corresponding Author:*
Lian Li,
SKL of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, University of Chinese Academy of Sciences, China;
Email: lianli@ict.ac.cn

request is decomposed into multiple subtasks, and asynchronous operations among different subtasks can interleave log messages differently. How to identify log messages for a particular subtask or user request is necessary, but challenging.

(2) Request flow spread across multiple systems. Cloud applications are built on top of multiple distributed systems, where each system may consist of hundreds of software components and thousands of nodes. For example, the cloud computing framework Hadoop2 consists of 3 sub-systems, the computing framework Hadoop Map-Reduce, the distributed resource management system YARN[12], and the distributed file system HDFS[4]. Hence, a simple user request can spread across multiple systems, involving hundreds of nodes and components.

An illustration Example



Application application_1520652966474_0001 is submitted by user user1
1s
Storing attempt: AppId: application_1520652966474_0001 AttemptId: appattempt_1520652966474_0001_000001
1s
Adding job token for job_1520652966474_0001 to jobTokenSecretManager
1s
10s
24s
task_1520652966474_0001_m_000000 Task Transitioned from NEW to SCHEDULED
task_1520652966474_0003_r_000000_0Task Transitioned from NEW to SCHEDULED
task_1520652966474_0001_r_000000_0 Task Transitioned from NEW to SCHEDULED
Task succeeded with attempt attempt_1520652966474_0001_m_000000_0
Task succeeded with attempt attempt_1520652966474_0003_r_000000_0
Task succeeded with attempt attempt_1520652966474_0001_r_000000_0
3s
5s
20s
job_1520652966474_0001 Transitioned from COMMITTING to SUCCEEDED

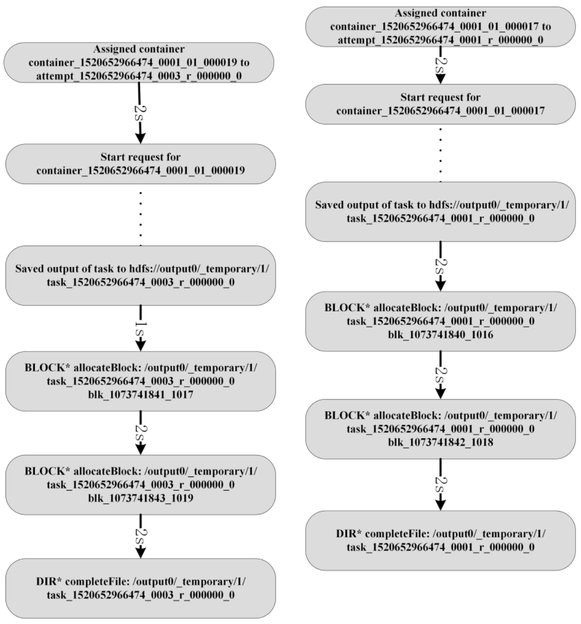
Figure 1. A simplified request flow extracted from Hadoop2/ YARN

Note: Each square represents an event, identified by a runtime log instance. Each edge is labelled with a corresponding latency to represent casual relation between events. In this example, the user submit a map/reduce job. As a result, a job attempt task is launched. The job attempt task is then divided into a set of sub-tasks. The job attempt succeeds when all subtasks finish and the job is finished.

Figure 1 illustrates a simplified request flow extracted by AgamottoEye from Hadoop2/Yarn [12]. Each square represents an event, uniquely labelled by a log instance. Events logging a same ID variable are grouped together to form a subtask, e.g., job_1520652966474_0001. A subtask may be further divided into sets of subtasks, e.g., task_15-20652966474_0001_m_00000… task_15-20652966474_0003_m_00000. The job finishes when all its three subtasks finishes.

Figure 2 shows the request flow of a particular subtask in Figure 1. Note that the events in Figure 1 are in Hadoop2 Map-Reduce/Yarn, and the events in Figure 2 are from a different system HDFS.

At last, Figure 3 presents a hierarchical view of the different tasks when processing a user request. Each task is associated with a unique ID variable, and values of ID variables are used to identify distinct task instances. Tasks are connected if their associated ID variables are printed in a same log statement. The ratio (1:1 or 1:n) denotes the ratio between number of task instances. For example, in Figure1, the same attempID value is printed in multiple log instances with distinct taskID values. Hence, the ratio between the two subtasks are 1:n, suggesting an attempID task instance is decomposed into multiple taskID task instances. On the other hand, the ratio between taskID and fileID is 1:1, suggesting a 1 to 1 mapping between the two tasks.



Assigned container container_1520652966474_0001_01_000019 to attempt_1520652966474_0003_r_000000_0
2s
Start request for container_1520652966474_0001_01_000019
Saved output of task to hdfs://output0/_temporary/1/ task_1520652966474_0003_r_000000_0
1s
BLOCK* allocateBlock: /output0/_temporary/1/ task_1520652966474_0003_r_000000_0 blk_1073741841_1017
2s
BLOCK* allocateBlock: /output0/_temporary/1/ task_1520652966474_0003_r_000000_0 blk_1073741843_1019
2s
DIR* completeFile: /output0/_temporary/1/ task_1520652966474_0003_r_000000_0
Assigned container container_1520652966474_0001_01_000017 to attempt_1520652966474_0001_r_000000_0
2s
Start request for container_1520652966474_0001_01_000017
Saved output of task to hdfs://output0/_temporary/1/ task_1520652966474_0001_r_000000_0
2s
BLOCK* allocateBlock: /output0/_temporary/1/ task_1520652966474_0001_r_000000_0 blk_1073741840_1016
2s
BLOCK* allocateBlock: /output0/_temporary/1/ task_1520652966474_0001_r_000000_0 blk_1073741842_1018
2s
DIR* completeFile: /output0/_temporary/1/ task_1520652966474_0001_r_000000_0

Figure 2. A simplified request flow of the subtasks in Figure 1

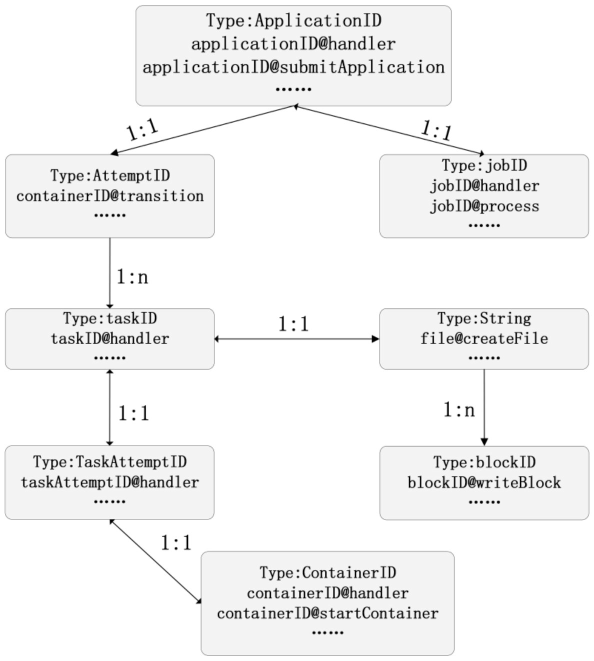


Figure 3. Relation between tasks in Figure 1

The above figures illustrate the detailed request flow of a service request at different granularity. They can be analysed manually for system understanding and profiling. Alternatively, such information can be combined with existing tools (e.g., Spectroscope[10]) to automatically detect anomalies.

2. Implementation

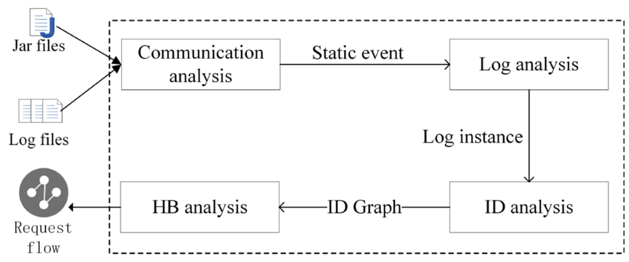


Figure 4. Overview of AgamottoEye

We implement AgamottoEye in Wala [8] via a series of sub-analyses (Figure 4). Communication analysis performs static analysis on the source code to identify these events sending messages to other components. These events will be handled by corresponding subtasks. Log analysis analyzes log instances and maps them to statically identified communication events. ID analysis groups events together into subtasks, according to the logged ID variable values. HB analysis computes the causal-relation between events using a customized happen-before model for distributed systems.

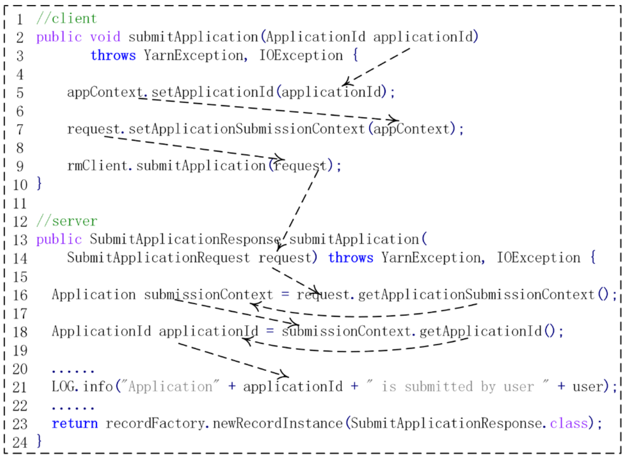
Casually-related events are connected together in the generated request flow.

(1) Communication analysis. We consider three types of communication events in distributed systems: thread creation, RPC (remote procedural call), and event dispatch. Each communication event is identified with a client side to send a request message, a server side method to handle the request, and a logging pattern encoded in regular expressions to log the event. For example, the logging pattern of the first event in Figure 1 is "Application (.*) is submitted by user (.*)"

(2) Log Analysis. Log analysis maps log instances to statically identified communication events, according to their logging patterns. ID variable values in log instances are extracted for further analysis. We use lucene[1] to speed up log analysis. As in Figure 1, for the first event, the logged ID variable values are "application_15206242966474_0001" and "user1".

We regard a variable as an ID variable if it is wrapped in the request, and printed by a log in the request handler method. Figure 5 gives an example. The client side set the ID value as a field of the request object (line #5-9 in Figure 5). The request handler method (server side) decomposes the request (its formal parameter), to get applicationID (line #14-18 in Figure 5), which is then printed in log statements (line #21 in Figure 5). Hence, AgamottoEye regards a variable as an ID variable if it is derived from formal arguments of a request message hander method, and is printed in log statements.

(3) Id analysis. ID analysis groups log instances together according to their associated ID variable values. Note that here we consider the logged ID values only, even if the log statements print different variables. As such, we avoid precisely analyzing the dependences between logged variables, to statically determine whether they refer to the same variable or not. Tasks are related if their associated ID values appear in a same log instance, as shown in Figure 3.



```
1  //client
2  public void submitApplication(ApplicationId applicationId)
3          throws YarnException, IOException {
4
5      appContext.setApplicationId(applicationId);
6
7      request.setApplicationSubmissionContext(appContext);
8
9      rmClient.submitApplication(request);
10 }
11
12 //server
13 public SubmitApplicationResponse submitApplication(
14     SubmitApplicationRequest request) throws YarnException, IOException {
15
16   Application submissionContext = request.getApplicationSubmissionContext();
17
18   ApplicationId applicationId = submissionContext.getApplicationId();
19
20   ......
21   LOG.info("Application" + applicationId + " is submitted by user " + user);
22   ......
23   return recordFactory.newRecordInstance(SubmitApplicationResponse.class);
24 }
```

Figure 4. Identify ID Variables

(4) HB analysis. With the task graph, AgamottoEye uses the logged values to map each log instance in the corresponding request flow. AgamottoEye computes the happens-before (HB) relation between log instances as follows: A. If the corresponding log points of two log message belong to the same task, the log points execution order determines the HB relation, B. if the log instance of one static communication event always occurs before the log instance of another static event at runtime, there exist HB relation between two static communication events.

After HB analysis, AgamottoEye creates nodes for each log instance. Nodes are connected if there exists HB relation between them. We refine the graph by removing transitive edges, and finally we have the request flow like the Figure 1.

3. Applications

We have used AgamottoEye to extract request flow automatically from Hadoop2/Yarn. It costs about 345.532

secs in total, to analyze 539,085 lines of code and 25,050 log instances. Most of the time are spent in analyzing the source code, including 52.52% of the time to build the call graph. Only 57.444 secs(16.62%) are spent in analyzing logs. This suggests that AgamottoEye is efficient enough to monitor request flow online, since only the new generated logs need to be processed.

In addition to generate the graphs (Figure 1 to Figure 3) for manual inspection, we also experimented whether the extracted request flow information can be helpful with automatic bug diagnosis. In this experiment, we use Spectroscope [10] to compare request flows generated from AgamottoEye, to detect anomalies.

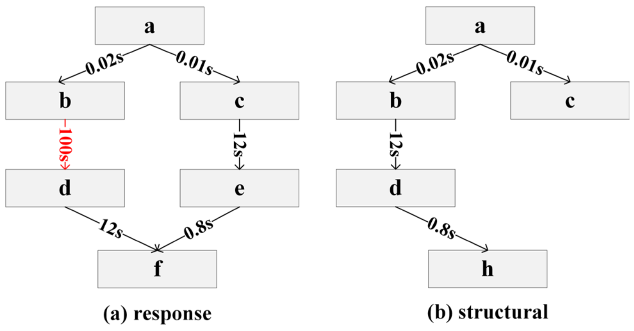


Figure 6. Two types of anomalies that Spectroscope can finds. The messages in (a) are the same as the normal work-flow. The messages of (b) are different.

Extend Spectroscope: Spectroscope detects anomalies by comparing request flows with the normal request flow (obtained via profiling). It can detect two types of anomalies: response anomaly and structural anomaly, as in Figure 6. Response anomaly is the case when the latencies on one or more edges become abnormally larger, e.g., the b->d edge in Figure 6(a). Structural anomaly denotes the case when some edges or nodes are added or lost in abnormal request, e.g., the node h and edge d->h in Figure 6(b). To detect structural anomaly, spectroscope classifies request flows into different clusters and compares two clusters to find abnormal structure edges. Spectroscope detect response anomaly by comparing request flows in the same cluster. Currently, Spectroscope uses depth-first search to get the string represent for each request flow, and request flows are classified according to their string representation. For example, the string representations for request flow in Figure 6 (a) and (b) are “abdfce" and abdhc", respectively. However, in our experiments, the generated request flow always have slight differenced even if we run the same workload in the same environment.

Hence, in our experiments, we use hierarchical clustering [7] to replace the origin clustering strategy. Our experiments shows that hierarchical clustering can effectively tolerate such slight changes and are able to detect anomaly with good precision.

Input: Our workload is WORDCOUNT of Hadoop2/YARN. We have run this workload for ten times in a three nodes cluster concurrently to generate normal requests. We also generate the abnormally request flows by : (1) randomly injecting sleep to simulate physical machine slowdown or network traffic delay. This will generate the response anomaly like Figure 6(a); (2) randomly injecting node crash events to simulate hardware failure. This will generate the structural anomaly like Figure 6(b); and (3) reproducing the bug MAP-REDUCE-3228[2] which will lead to request hang. This will generate both response and structural anomaly.

Result: AgamottoEye can successfully construct the request flows for all above inputs. We have compared the three types of abnormal request flow with the normal request flows using our extended Spectroscope. For the first type, Spectroscope can correctly point out which edge becomes slow. For the second type, Spectroscope can successfully identify recovery and missing edges introduced by node crash. For the third type, Spectroscope can pinpoint the three edges related to the bug. The experimental results demonstrate the precision of request flows generated by AgamottoEye.

4. Related Works

Sambasivan et al. [9] summarize how to generate request from from end-to-end tracing techniques. The tool lprof [15] generates the request flow for each thread. Stitch [14] maps all logs to their corresponding request flow, but did not compute the causal relation between them. Cloudseer [13] uses an automaton to depicts a task workflow, which is built from existing logs and can be used to monitor request status online. AgamottoEye differs with the above work and automatically generates the request flow across multiple systems.

5. Conclusion

In this paper we propose AgamottoEye, a new tool to recover request flow from existing logs. AgamottoEye precisely analyzes interleaved log instances and can process request flow spread across multiple distributed systems. Our experimental results show that the generated request flow can help developers to diagnose and detect anomalies.

References

[1] http://lucene.apache.org/
[2] MR AM hangs when one node goes bad.
https://issues.apache.org/jira/browse/MA-

PREDUCE-3228

[3] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling.. In OSDI, 2004, 4: 18.

[4] Dhruba Borthakur et al. HDFS architecture guide. Hadoop Apache Project, 2008, 53.

[5] Yen-Yang Michael Chen, Anthony J Accardi, Emre Kiciman, David A Patterson, Armando Fox, and Eric A Brewer. Path-based failure and evolution management, 2004.

[6] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In Proceedings of the 4th USENIX conference on Networked systems design & implementation. USENIX Association, 2007, 20–20.

[7] Anil K Jain and Richard C Dubes. Algorithms for clustering data, 1988.

[8] T.J. Watson Libraries.
https://github.com/wala/WALA

[9] Raja R Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R Ganger. So, you want to trace your distributed system? Key design insights from years of practical experience. Technical Report. Technical Report, 2014. CMU-PDL-14.

[10] 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 NSDI, 2011, 5: 1.

[11] 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. Technical report, Google, Inc, 2010.

[12] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 2013, 5.

[13] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. In ACM SIGPLAN Notices, 2016, 51. ACM: 489–502.

[14] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle.. In OSDI. 2016, 603–618.

[15] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems.. In OSDI, 2014, 14: 629–644.