Capturing Request Execution Path for Understanding Service Behavior and Detecting Anomalies without Code Instrumentation

Yong Yang, Long Wang, *Senior Member*, *IEEE*, Jing Gu, Ying Li, *Member*, *IEEE*

Abstract— With the increasing scale and complexity of cloud platforms and big-data analytics platforms, it is becoming more and more challenging to understand and diagnose the processing of a service request across multi-layer software stacks of such platforms. One way that helps to deal with this problem is to accurately capture the complete end-to-end execution path of service requests among all involved components. This paper presents REPTrace, a generic methodology for capturing such execution paths in a transparent fashion. Moreover, this paper demonstrates the effectiveness of REPTrace by presenting how REPTrace can be leveraged for knowledge extraction and anomaly detection on the platforms' request processing. Our experimental results show that, REPTrace enables capturing a holistic view of the request processing across multiple layers of the platforms (which is missing in official documentation) and discovering important undocumented features of the platforms. Fault injection experiments show execution anomalies are detected with 93% precision and 96% recall with aid of REPTrace.

Index Terms— end-to-end tracing, service request, knowledge extraction, request execution path

1 Introduction

Many services and applications are now hosted by cloud platforms such as AWS, MS Azure, SoftLayer, Openstack. Besides traditional services, these services/applications include analytics ones such as AWS machine learning services [1], IBM Watson Healthcare Consulting services [2], and Google Cloud Bigquery services [3], which run on analytics platforms such as Hadoop, Spark, Tensorflow, and Jupyter Notebook. These services/applications run on multi-level software stacks in the platforms. Expertise on such platforms, software stacks and hosted services is critical for the enterprises of platform providers such as Amazon, IBM and Google, to maintain and improve the platforms and services.

However, it is demanding for the platform provider enterprises to obtain and keep the up-to-date expertise on the platforms, software stacks, and services due to the following reasons in our problem context: i) new services and applications continue to onboard the platforms; ii) software of such platforms, software stacks, and services have been developed by different vendors or teams and may be owned by different parties, and source code of them may be unavailable; iii) the system software and the services are under continuous and rapid updates (e.g. OpenStack is released every 6 months [4]), iv) there is frequent flow of personnel in operation/development teams (this makes the knowledge transfer a headache for most large providers of computing platforms). Actually, it is reported that up to 60% of the software engineering effort is spent on understanding the software system [5].

Therefore, the idea of extracting knowledge from system behavior itself is quite attractive to large enterprises of computing platform providers. Drawing message flows of individual request's processing among the components of a distributed platform was demonstrated to be an effective way to understand and analyze behavior of cloud platforms. For instance, the analysis of OpenStack's release evolvements [6] presented manually drawn message flows among OpenStack's components for identifying behavior changes in different OpenStack releases.

However, manual construction of message flows or other capturings of system behaviors is inefficient towards the goal of extracting up-to-date understanding/knowledge on such a platform and hosted services. Moreover, we intend to *capture the complete end-to-end execution path of processing an individual request among all involved components of the platform.* We believe the complete path conveys a holistic view of the platform's behavior during the processing of the request in the software stacks, and this holistic view, with in-depth details, boosts the up-to-date understanding/knowledge and even diagnosis of request processing behavior, especially concerning the reasons described above. This capturing of the end-to-end path in our specific context truly differentiates our work from previous ones, and is attractive to enterprises providing such platforms like IBM Watson Health Cloud [2].

A few *research challenges* should be addressed for capturing the complete end-to-end execution path for a service request: i) a thread may be servicing multiple requests concurrently, and we should identify the execution segments (at a fine granularity such as system calls) of the thread's processing of each individual request in presence of the multi-request concurrency and link these segments

- *Yong Yang is with Peking University, Beijing, China, 100871. E-mail: yang.yong@pku.edu.cn*
- *Long Wang is with Tsinghua University, Beijing, China, 100084. E-mail: longwang@tsinghua.edu.cn*
- *Jing Gu is with Peking University, Beijing, China, 100871. E-mail: gu.jing@pku.edu.cn*
- *Ying Li is with Peking University, Beijing, China, 100871. E-mail: li.ying@pku.edu.cn*

into a complete execution procedure of the thread correctly; ii) the processing of the request traverses multiple threads, processes, components, and even computer nodes in many scenarios of service computing, so we should identify the execution segments of this request's processing among these distributed parts and link them correctly; and iii) we should address the previous two challenges in situations when the source code is unavailable.

A number of technologies in current literature try to tackle similar execution-tracing problems, e.g. Stardust [7], Xtrace [8], Dapper [9], Pinpoint [10], Pivot [11], Magpie [12], HDFS HTrace [13] and Facebook's Canopy [14]. These technologies either require vendor-specific libraries (e.g. Google's own RPC libraries), or deal with specific network protocols only (so unable to identify and link intra-thread execution segments of a request's processing at the fine granularity), or instrument source code of the involved software, for enabling the tracing capabilities. These existing technologies do not apply for our challenges above.

A project close to our work is vPath [15]. This project traces communications of distributed systems/applications by pairing each network message's sending (by the sender thread *S*) and receiving (by the receiver thread *R*) via the TCP/IP connection information. Because vPath only relies on pairing of TCP/IP connection information for identifying causality, this makes vPath work only in one synchronous communication/thread pattern: a) *S* sends a message to *R* via connection *c* and blocks itself; b) if *R* replies to *S* via connection *c* then *S* resumes its execution (and R stops working and waits for receiving another message), or if *R* sends a message to another thread *T* via another connection d, R blocks itself until it receives reply from *T* via *d*. For any other communication pattern or thread pattern, vPath will fail (as the vPath authors stated in [15] and our experiments also showed). These other patterns, which are common in service computing, include asynchronous communication, thread forking or thread handover during request processing, connectionless communication (UDP), connection pool, event driven and thread pool. We needed a general mechanism that traces distributed platforms' processing of service requests covering most, if not all, service computing scenarios. vPath was the closest work we found after we searched the literature, but soon we found it did not work for our purposes due to the pattern constraint.

In this paper a Request Execution Path (REP as acronym) is defined as the complete path of processing a specific service/job request. Specifically, the REP 1) covers all of the executions of the given distributed platform's components in processing a specific request (at a given granularity such as library/system calls), including executions of those components' processes and threads as well as the communications among them during the processing; 2) identifies all these executions, and links them together according to the accurate causal relationships among them to form an integral unseparated view of the platform's processing.

We propose Request Execution Path Trace (REPTrace) to capture the REP. REPTrace intercepts runtime events such as common library/system calls at the operating system level (either within a physical machine, VM, or container). Principles and algorithms of REPTrace identify the relationships among the events. Then REPTrace stitches all the events of one request's processing together using the identified relationships, and produces the complete REP. To address the research challenges described above, in REPTrace we i) perform comprehensive analysis of execution scenarios about how distributed systems process requests, ii) modify network messages with extra information for pairing the sending and receiving of the same message, and iii) maintain the context of each request's processing to differentiate it from other concurrent requests. REPTrace intercepts a set of relevant library/system calls for these operations and do not require the software's source code.

As a demonstration of REPTrace usage, this paper also presents an anomaly detection approach that leverages REPs to detect anomalies during processing of a request, as well as our experience of extracting knowledge on distributed system behavior through REP of individual service/job request's processing. In summary, this paper makes the following contributions (actually this journal paper is based on our previous two conference papers [16] and [17], and we explain what contents are in the two published papers below):

- We analyze execution scenarios of request processing in computing platforms and identify event relationships in all these execution scenarios. This analysis is the basis for REPTrace, and is one of the main contributions of our conference paper [16].
- We propose REPTrace for constructing complete REPs without code instrumentation (by "code instrumentation" we mean the modifications of relevant software source code directly), and implement REPTrace on Linux systems. Novel algorithms are devised for the REP construction and no specific communication or thread patterns are assumed in REPTrace. As far as we know, there is no prior art of distributed tracing that tries to address the comprehensive communication and execution scenarios of service request processing as we handle. The very high-level idea of the REPTrace was described in [16], but the algorithms and the crystallized REPTrace are only presented here.
- An anomaly detection approach based on REPTrace is devised to demonstrate how REP can be leveraged to detect execution anomalies by discovering deviations of the execution paths from normal ones. We evaluate REPTrace and the anomaly detection method with extensive experiments in terms of performance overhead, anomaly detection precision, recall and F1-measure on different distributed platforms. Our experimental results show that with a 4.5% execution latency overhead the REPTrace-based anomaly detection approach detects anomalies with 93% precision and 96% recall.
- We also present our experience of using REPTrace to automatically extract knowledge on distributed system behavior of request processing without source code or prior knowledge. Our experimental results show that, with the aid of REP our approach is able to deliver a ho-

listic view of the request processing across multiple layers of distributed systems (the holistic view is missing in official documentation of the systems in our experiments). The knowledge extraction is the main contribution of our conference paper [17].

- We performed extensive experimental evaluations of REPTrace, including the path analysis, overhead analysis, the evaluation of anomaly detection, and the comparison with other leading anomaly detection approaches. The comparison shows that our REPTrace-based anomaly detection outperforms leading anomaly detection approaches such as DeepLog [37] and MRD [38] in that REPTrace provides a global holistic view of request processing, including both the interactions among different microservices and those within each single microservice.

The rest of the paper is organized as follows. Section 2 presents an analysis of different scenarios of service request processing and introduces how REPTrace captures the complete request execution paths in these scenarios. Section 3 describes essential data processing of REPs. An anomaly detection method based on REPs is presented in Section 4. The experimental evaluation is given in Section 5. Section 6 analyzes the related work and Section 7 concluds the paper.

2 REP Construction

Here we list 7 scenarios involved during service request processing based on our experience and analysis of real-world IT paltforms. We believe these scenarios are comprehensive and cover most, if not all, of the common execution scenarios in service request processing (they cover all cases of service request processing in our experiments). Figure 1 illustrates these scenarios. We analyze this comprehensive list for supporting general request execution rather than limited patterns only.

a) Continuous execution within a thread;

b) The current thread creates another thread and passes the handling of the request to the new thread. The current thread may stop processing (e.g. sleep), or continue processing this request;

c) The current process forks a process, and passes the handling of the request to the new process. The current process may stop or continue processing this request;

d) The current process/thread sends a message to another process/thread, which may be on the same machine or a different machine. Then the latter process/thread begins the processing of the request. Network communication and other similar mechanisms, like pipe, are covered in this scenario. The current process/thread may stop or continue processing this request;

e) The current process/thread synchronizes the processing of the request with another existing process/thread using certain IPC (Inter-Process Communication) mechanism such as process wait, thread join, signal, lock/unlock, semaphore, etc.;

f) The current process/thread saves the request (or its intermediate state) in a message queue. Then a different process/thread picks up the request (or the intermediate state) from the message queue and begins processing;

g) The current process/thread passes the handling of the request to another existing process/thread using shared memory, shared variables, or mapped device.

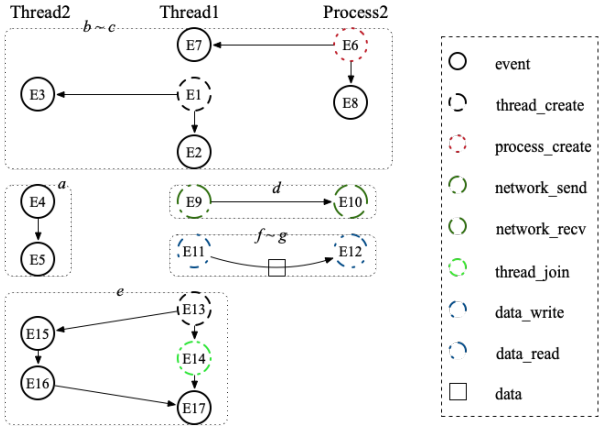


Figure 1: Different categories of execution scenarios

By looking into the service request execution (and the scenarios above), we see the execution consists of the following sequence: … - *event* – *thread execution* – *event* – *thread execution* - …, where *event* indicates a system call (or a LIBC call encapsulating a system call) that performs a relevant operation, and *thread execution* indicates one thread's continuous execution between one event and its successive event. After removing *thread execution* from the sequence, the execution process is then denoted as the sequence … - *event* – *event* - …. In this paper we use such sequences of events to denote the execution of request processing. Suppose an event *e1* has its immediate successive event *e2* in the sequence; we call e1 predecessor (*pred*) of *e2* and *e2* successor (*succ*) of *e1*, and their relationship is denoted as *e1->e2*. Note that one event may have multiple *succs* or *preds* in certain scenarios (see more in Section 2.2).

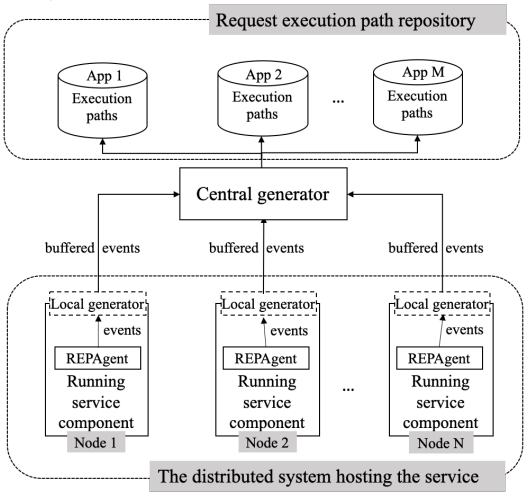


Figure 2: System architecture of REPTrace

REPTrace Architecture. REP construction involves generation of events and linking of events. Figure 2 shows the architecture of REPTrace. A REPAgent on each node (deployed inside a container, a virtual machine, or a physical machine) of the traced distributed system intercepts LIBC calls that encapsulate those system calls of interest and generates events. The events are buffered locally in Local Generator as they are generated and then uploaded to the Central Generator in chunks when the buffer is full. If the buffer is set to hold only one event, each event is emitted to Central Generator when it is generated, which

supports online analysis. The Central Generator collects these events, conducts the event linking, and then stores the complete REP of every request in the repository.

2.1 Event Generation

According to the listed 7 scenarios (*a~g*), the events of our interest fall in one of the following types: thread/pro-cess creation, network communication via messages, thread/process synchronization and thread/process interactions via messages queues and shared objects. Therefore, we identify the relevant LIBC calls in the following 5 categories and intercept them for event generation: thread manipulation, process manipulation, network communication, synchronization, and other, as listed in Table 1.

REPAgent intercepts LIBC calls of the node via the LD_PRELOAD mechanism. Upon an interception, REPAgent generates a single event, aka. trace event, besides invoking the original LIBC call. The trace event has attributes that are common for all categories of events, such as *event ID, call info, thread info, process info, node info, local time info, event linking info*. The *event ID* is a UUID (Universally Unique Identifier) generated by REPAgent with the libuuid library [40]. Here *call info* includes the name of the intercepted LIBC call as well as the parameters and return value of the call; *thread info* has the thread ID of the intercepted call; *process info* has the process ID of the intercepted call; *node info* has the ip address where the interception occurs; *event linking info* includes attributes our algorithm creates or uses to link events and form REP. These attributes are either generated by REPAgent or extracted from corresponding data structures which are passed into the intercepted calls as parameters. The trace event also has category specific attributes such as *port number*.

Table 1: LIBC calls intercepted for event generation

Category	Examples
Thread manipulation	pthread_create, pthread_join, pthread_self, pthread_detach, pthread_cancel
Process manipulation	fork, vfork, exec
Network communication	send, recv, write, read, sendmsg
Synchronization	wait, waitpid, pthread_join, signal
Other	open. close, malloc, syscall

2.2 Event Linking

All of the generated trace events are linked to form the REP via their pred-succ relationship. So, the key of event linking is to identify the pred for each event. The pred-succ relationships of the generated events in the above-listed execution scenarios are summarized as five types.

The following discusses the five types and how we identify the predecessor (*pred*) of an event associated with the event relationship types. For sake of presentation convenience, we describe Type 3 first. One event may have multiple preds if more than one rule applies. Algorithm 1 describes the event linking algorithm for the types of 1~4, which is executed by the Central Generator. The linking for type 4 is lines 3-4, for type 3 is lines 7-9, for type 1 is lines 11-13, and for type 2 is lines 14-21. The linking for type 5 (data flow of the request processing among threads/processes) is not shown in Algorithm 1, but is also performed together with the operations for types 1~4. The event linking can be performed in both an online mode and a postmortem mode. In the online mode the event linking algorithm is used to process event streams and identify the parents of the newly generated event.

```
Algorithm 1 Event_Linking
Input: trace events set S
Output: parent[e] for each event in S
1.  For each event e in S :
2.     parent[e] = GetParent( e )
3.     If parent[e].call_id in [ thread_join, wait, waitpid ] : // Type 4
4.        assign parents to e based on the call's specific semantics
5.
6.  GetParent( e ) :
7.     If e is of recv event and e.MSG_ID not empty : // Type 3
8.        find event p in S for p.MSG_ID = e.MSG_ID
9.        Return p
10.    else
11.       find event p in S for p.MSG_CTX_ID = e.MSG_CTX_ID  // Type 1
                  and p.thread_id = e.thread_id
                  and p.time_info is closest before e
12.       If p is found :
13.          Return p
14.       /* this means e is the first event of its thread/process */
15.       find event p in S for p.MSG_CTX_ID = e.MSG_CTX_ID // Type 2
                  and p.call_id = thread_create
                  and p.return_value = e.thread_id
16.       If p is found :
17.          Return p
18.       /* this mean E is the first event of it's process */
19.       find event p in S for p.MSG_CTX_ID = e.MSG_CTX_ID // Type 2
                  and p.call_id in [fork, vfork]
                  and p.return_value = e.process_id
20.       If p is found :
21.          Return p
22.       /* this means e is the staring/root event of a REP */
23.       Return NULL
```

• *Type 3: Causality of communication between threads/processes* in scenario *d* (*e9->e10*). The communication events can be classified into two types, *send* and *recv*. The *send* event in the sender thread is the pred of the corresponding *recv* event in the receiver thread.

REPAgent inserts a unique ID, called MSG_ID (part of the MSG_ID is a UUID to ensure the uniqueness), into each transmitted message of the intercepted *send* call, and then REPAgent on the receiver side extracts this MSG_ID from the received message of each intercepted *recv* call. This MSG_ID is one attribute of the *event linking info*. For any *recv* event, its pred is the *send* event with the same MSG_ID. Unlike vPath which uses TCP connection and socket information to pair *send* event and corresponding *recv* event, our mechanism of message labelling does not rely on specific communication pattern or thread pattern for doing the pairing.

Implementation Details. The MSG_ID includes the length information of the message to deal with TCP message fragmentation. When a message sent by one *send* event is fragmented and received by multiple *recv* events, or messages sent by multiple *send* events are received in one or multiple *recv* events, REPAgent extracts the length information upon the interceptions of *recv()* calls, and exploits the length information to identify message borders and merge/split the received fragments into original messages transmitted at *send* events. As a result, one *send* event maps to one *recv* event with the same MSG_ID.

• *Type 1: Temporal successiveness within the same thread* in scenario *a* and other scenarios (e.g. in asynchronous messaging mode a thread continues processing of the request after sending message). Examples are *e4->e5*, *e1->e2*, *e6->e8*, *e13->e14->e17*, *e15->e16* in Figure 1. Only those events that participate processing of the current request are considered, and the events of the same

thread's processing of other requests in concurrency should be excluded for the REP generation of this request.

The *thread info*, *process info*, *node info* of events are exploited to find those events generated within the same thread of the given event. Then *local time info* is exploited to identify which event has the time immediately ahead of the given event, i.e. the pred event.

One challenge is to exclude those events that do not participate processing of the current request. For instance, in the thread pool pattern a thread had been created long before the current request was initiated, and those events before the current request should be excluded. In another example, a thread may be serving multiple requests concurrently, and we should exclude those events of processing other requests.

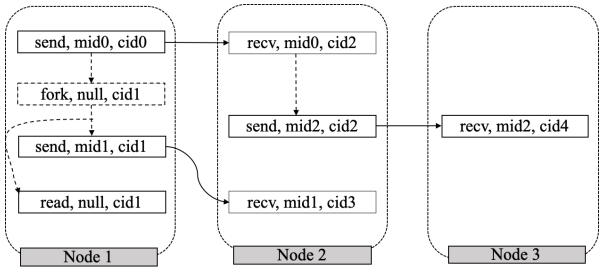


Figure 3: An example of MSG_ID and MSG_CTX_ID propagation among events (each tuple in the figure is <*call ID*, *MSG_ID*, *MSG_CTX_ID*>, a simplified version of the event)

We define a unique ID, MSG_CTX_ID (part of it is a UUID to ensure the uniqueness), to aid maintaining the request's context and excluding irrelevant events. The intuition behind the MSG_CTX_ID concept is to group those local events that are related to the same individual message. As the initial service/job request received by the computing platform is a message, the grouped events by this message's context are related to processing of this request. If one of the grouped events is a *send* event, its corresponding *recv* event is identified by the MSG_ID. Upon this *recv* event a new message context starts with a new MSG_CTX_ID; the *send* event also starts a new message context within its own thread. Therefore, the basic idea is to leverage the cascading use of MSG_CTX_ID and MSG_ID for grouping all events related to the original request. MSG_CTX_ID is also an attribute of *event linking info*. Figure 3 gives an example of MSG_ID and MSG_CTX_ID propagation along events during request processing (REPAgent puts MSG_ID and MSG_CTX_ID into the events submitted to the Central Generator). Note that MSG_CTX_ID may propagate across creations of process/thread (the *read* in Node 1 is the first event in a newly-forked process).

Implementation Details. More strictly, MSG_CTX_ID is defined as a unique identifier of the continuous computation from the sending or receiving of one message (event A) to the sending or receiving of the next message (event B). All events between event A and event B along the request execution have the same MSG_CTX_ID. If event A is a *recv* event, a new MSG_CTX_ID is created and marked for both event A and those events after A (and before B); if event A is a *send* event, the event A is still marked with the old MSG_CTX_ID for purpose of event stitching, and a new MSG_CTX_ID is created for those events after A (and before B). The MSG_CTX_ID of event B is similarly assigned as there is a next message after B.

Each thread has a current MSG_CTX_ID. REPAgent maintains a table of the current MSG_CTX_IDs for all involved threads in the node. On intercepting a message-sending or message-receiving call, REPAgent updates the thread's current MSG_CTX_ID with a new value and marks the event's MSG_CTX_ID with either the old value or the new value according to the description above. When an event other than a *send* or *recv* event is generated, the event's MSG_CTX_ID is assigned as the thread's current MSG_CTX_ID value (a current MSG_CTX_ID is created for the thread by REPAgent if it does not exist yet).

- *Type 2: Causality of thread creation and process creation* in scenarios *b* and *c* (*e1->e3*, *e6->e7*, *e13->e15*). In this type of relationship, the first event in the new thread/process is the succ event of the thread-creation/process-creation event in the parent thread/process. The thread/process-creation event's *call info* has the ID of the newly created thread/process returned by the call. So, if an event is the first event of a thread/process, we search those thread/process-creation events and identify its pred. MSG_ID and MSG_CTX_ID are used to exclude irrelevant events as well.
- *Type 4: Causality of synchronization among threads* in scenario *e* (*e16->e17* where *pthread_join* is the synchronization method). The event linking depends on the specific synchronization mechanism. For each of the synchronization mechanisms, we define its specific causality identification method. For example, *waitpid()* blocks the calling process A (the current process) and waits for the specified process B to terminate or change state (stopped or resumed or signaled). Then process B's last event before B's termination or state change is the pred of process A's event that is immediately after the *waitpid()* in the control flow order. The *waitpid()* event of process A is also the pred of this process A's event (multi-pred). Similarly, specific identifications of event causal relationships associated with thread joining, signals, lock/unlock and other IPC mechanisms are defined correspondingly based on their semantics in manipulating execution behavior.
- *Type 5: Causality via data dependency between threads/processes* in scenarios *f* and *g* (*e11->e12*). The data dependency is *read-after-write* dependency between threads/processes. The event that writes data in one thread is the pred of the event that reads the written value in another thread. If the write/read of the data is not via a system call but via direct memory access (e.g. shared memory, shared variable), the event that is immediately before the *write* is the pred event, and the event that is immediately after the *read* is the succ event.

The event relationship types 1~4 (scenarios *a~e*) are about manipulations or interactions of system control objects, i.e. thread, process, signal, communication, etc.; they indicate control flow of the service request processing among threads/processes. The type 5 (scenarios *f* and *g*) indicates data flow of the request processing among threads/processes.

General transparent accurate analysis of data depend-

ency relationships along individual traces is a difficult problem to tackle. This paper mainly addresses the construction of REP using control-flow relationships rather than data-flow relationships, as handling data dependency relationships is a radically different topic than handling control-flow relationships. However, we are still able to identify data dependency relationships in certain situations, by leveraging certain IDs which are typically available in many standard queue protocols and middleware. Such IDs include message ID, job ID, session ID, etc. For example, in Apache ActiveMQ there is a unique message ID in every message header; every job being submitted and executed in Hadoop and Spark has a job ID. The data writing and reading events can be correlated by these data IDs. We only consider the data dependencies in the relationships of type 5 for the event linking purpose, and do not attempt to address general data dependencies. Our implemented REPTrace tool only identifies the relationships of type 5 when such IDs are present.

If we do not address the data-induced causality between threads/processes (e.g. message queues) during event linking, the achieved REP is not a single complete path, but comprised of several path segments. In our experiments REPTrace reports 6 paths for a single job's processing when we do not consider such causalities, because job queues or message queues are used in Hadoop. After we leverage the Hadoop job ID and RPC ID to address this type of causality, REPTrace produces one execution path for one job request, as we expected.

2.3 REP Representation

A REP is built for one service request's processing. As an event may have multiple preds, the formed REP is denoted as a graph. Each vertex of the graph denotes an event and each edge denotes an event relationship. Each vertex has zero or a number of succ vertexs. All the generated events during processing of the request, from the request entering the service-hosting distributed components until final results for the request being replied, form the complete REP graph.

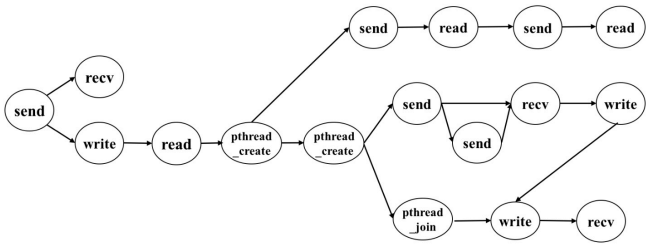


Figure 4: Example REP Fragment

The linkings of the types 1~4 event relationships involve either local successiveness information (e.g. local timestamp within a thread) or matching of certain values (process id, thread id, MSG_ID, etc.), so these linkings are unidirectional. There might be inconsistency among reads/writes of the same variable/memory location in Type 5 event relationship if the traced software has unresolved race conditions. If such an inconsistency occurs the REP graph might or might not contain a cycle. But we do not see any cycle in our experiments, maybe because either the probability of activating race condition bugs of the traced software (they are widely used mature software), or the probability of such inconsistency resulting in a cycle in the REP, is quite low. Therefore, the REP is represented as directed acyclic graph (DAG) starting with the receiving of a single request as the root event. Figure 4 shows an example of a REP fragment (only the event type is shown and other event information is omitted for the illustration clarity).

3 Essential Processing of REPs for Knowledge Extraction

Constructed REPs can be employed for helping understand service behavior in the application scenarios such as feature study, anomaly detection and knowledge extraction. There are certain essential data processings of REPs for the application scenarios as illustrated in Figure 5. The operations include data pre-processing, component identification and component communication identification. Here we give a brief discussion of them.

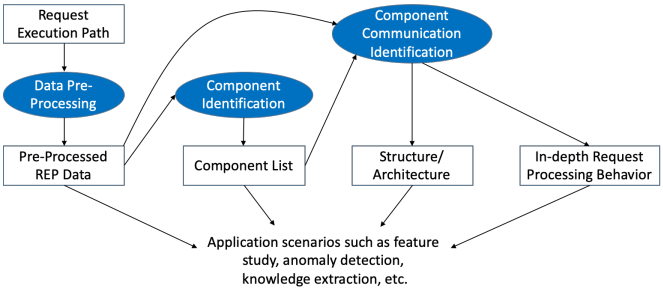


Figure 5: Essential data processing of REPs for application scenarios

3.1 Data pre-processing

Arbitrating Multi-Pred to Single-Pred. Certain events have multiple preds when two or more rules in Section 2.2 apply. We arbitrate the multi-pred linkings to single-pred, which converts a DAG into a tree of events, for facilitating our loop/concurrency pruning algorithm (it deals with trees). The arbitration is done by selecting only one pred relationship for any multi-pred event and removing the other pred relationships. We propose a subjective priority order (from highest priority to lowest) in deciding which rule to observe when competing with the other rules:

Rule on Type 3 > Rule on Type 5 > Rule on Type 4 > Rule on Type 2 > Rule on Type 1.

The same prioritization is observed for all cases; so, the arbitration is deterministic. We select this arbitration principle based on our understanding of typical impacts of the relationship types on request processing behavior.

Event Consolidation for Pruning Loop/Concurrency. There are many iterations and concurrencies in processing of a service request. One important task in our data preprocessing of the request execution path is to consolidate trace events so that the identical patterns of events are represented by only one copy for the further analysis. Identical pattern means two sequence fragments of events are same in terms of the number of events in the fragment, the order of the events in the fragment, the function types and the process IDs of those events in same positions of the fragment. Identical patterns are largely caused by iterations and invocations of same functions by differ-

ent threads. We exploited a classical string-pattern algorithm in [46] to do the pruning, which we later find is close to the iteration pruning in SpecMiner [43]. The consolidation largely reduces the number of events and simplifies further analysis (the range of the iteration numbers of an iterative substructure is recorded for future use).

After the event consolidation, the event tree changes into a graph with cycles (loops now represented as cycles).

3.2 Component Identification

This operation module identifies the list of components of the distributed system in processing the given request. Component identification is done because a component may have multiple processes/threads, and for human understanding of a system it is highly preferred to understand interactions at the component level rather than the process/thread level. The pre-processed REP is first converted to a process tree (or forest), and then the components are identified from the process tree. Figure 6 shows the procedure of component identification. The number of each node in a process tree represents process ID. The conversion from the event tree to the process tree is done by removing from the tree all those events not related to process manipulation. The result process tree depicts the parent-child relationship of processes.

The set of processes in the process tree (or forest) is the output of the component identification module, with each process regarded as a component. For charting neatness, in cases with many processes in the process tree and when people may get confused, the component identification module selects those processes with the tree depth less than a certain value so that the number of selected processes is not too many (e.g. less than 15). Figure 6 shows there are 2 levels in the process forest: tree depth 0 and tree depth 1, and only the first-level processes are considered as components. Each identified component has the name assigned as the combination of the process name (available in the trace event vector) and the process ID.

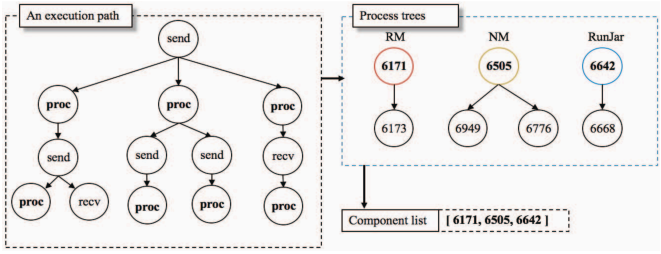


Figure 6: Illustration of component identification procedure

3.3 Component Communication Identification

As Figure 5 shows, this operation module takes both the pre-processed data and the identified component list as the input, and creates an overall view of the static structure/architecture of the distributed service components in request processing (structure view) and a view of the in-depth interactions of the components along the request processing (dynamic behavior view), from the beginning of the REP until the end of the path.

For each event in the pre-processed data, we mark which identified component it is associated with. Then each edge e1->e2 in the event tree is represented as a vector <e1's component, e2's component, e1's time, e2's time>, and the entire pre-processed event data are represented as a whole set of such vectors.

Generating dynamic behavior view. Sort all the vectors according to the time information, and then draw the dynamic behavior view. Figure 7a) illustrates an example dynamic behavior view for a WordCount job running on Hadoop platform in a given time window. The horizontal axis indicates the component names and the vertical axis indicates the timeline. The view shows the communications among the components in chronological order from top to bottom (only four communication pairs are shown in Figure 7).

Generating structure view. This static structure is drawn based on the set of vectors with the time information discarded. The Figure 7b) shows that the communications within processes/threads of the same component are not drawn in the structure view.

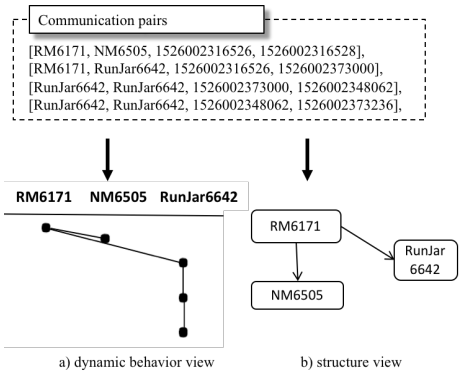


Figure 7: Generation of dynamic behavior and structure views

4 REPTrace-based Anomaly Detection

REPTrace can be exploited for anomaly detection of the service request processing. The REptrace-based Anomaly Detection method, READ as the acronym, is composed of two stages: training stage and detection stage. The basic idea is that, a couple of finite state automatons (FSAs) are constructed from the REP during the training, and then the runtime behavior of a request processing is compared against the FSAs to detect anomalies.

4.1 Training Stage

In this stage we run a number of requests of the same type concurrently on the service platform, collect the REP for each of the requests, generate a per-path FSA for each collected REP, and then merge these per-path FSAs for all the requests of this type, into two aggregated FSAs.

By request type we mean the function of a service which is requested to be performed. For instances, provisioning of a VM, provisioning of a container, shutdown of a VM, migration of a VM are different types of requests supported by a cloud platform. In a platform with REST API exposed, usually different request types have different URLs or different request modes; in an example command "*curl -ku user:passwd -X POST https://ip:port/servicename/api/v1/the_url -d @jsonobj*", *the_url* and the request mode POST determine the type of this example request. In a service platform like a production cloud, service requests are typically REST requests and service/cloud providers are able to identify the types of ser-

vice requests (the URL and the REST mode) at the service entrance interface automatically where the requests are received, according to our experience in IBM Watson Health Cloud. In a job processing platform like Hadoop, different applications on Hadoop are regarded as different types of job requests.

The two aggregated FSAs will be employed to detect anomaly of the execution of this type of requests in real workload during the detection stage. We repeat this training for all types of requests of interest supported by the service platform. We aggregate FSAs for each request type rather than for all request types because the processing logics of different types of requests vary a lot (e.g. a VM-provisioning request is completely different from a VM-shutdown request), building a big FSA to tolerate their variations while still provide good detection accuracy has unnecessary complexities, and the accuracy may be sacrificed, too. As the request type information is known at the beginning of request processing, it is straightforward to do anomaly detection for each request type.

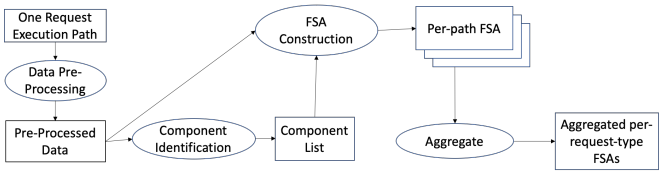


Figure 8: The Process of Generating a Per-path FSA

4.1.1 Generating a Per-Path FSA

Figure 8 illustrates the process of generating a per-path FSA from one request's REP and then aggregating all per-path FSAs of the same request type into the aggregated FSAs. The data pre-processing and component identification steps are same as explained in Sections 3.1 and 3.2.

A per-path FSA is constructed directly from the consolidated REP via a graph transformation: each vertex of the consolidated path (group of consolidated events) transforms into an edge (transition) of the FSA, and each edge of the consolidated path transforms into a vertex of the FSA (state between two groups of consolidated events). A state *St0* is added before the first transition (i.e. root event) of the FSA as its start state. An FSA state with an out-degree of 2 or more indicates either a branching scenario (conditional statement or loop) or a concurrency. Same as the pre-processed REP, the FSA contains cycles, which can match different numbers of repetitions of same event patterns during the detection stage.

4.1.2 Aggregating FSAs for a Request Type

We observe that normal execution paths of the same type of requests may be slightly different due to various causes such as retries after failures, built-in non-determinisms in the service platform, etc. Therefore, using an FSA from only one REP for anomaly detection suffers from a large number of false alarms. We address this issue by combining all collected per-path FSAs of a request type to generate two aggregated FSAs, a core FSA and a full FSA, for anomaly detection.

The core FSA is the intersection of all the per-path FSAs, and the full FSA is the union of all the per-path FSAs. Thus, the core FSA represents the common execution path fragments/state transitions for a request type, and the full FSA represents all the possible execution paths/state transitions for all REPs of a request type. The combining process is illustrated in Algorithm 2. Compared with another FSA-merging algorithm kTail [41], which merges two FSAs and their states based on whether two states' future k consecutive operations are the same, our Algorithm 2 decides whether two states from two FSAs can be merged based on the traversed states from their initial state *St0* to each of the two states (called *state_path*). If the two states have the same *state_path*, they represent the same state in the two FSAs and can me merged into one state in both the core FSA and the full FSA (the ***transition_path(S)*** function in Algorithm 2 returns the *state_path* for a given state S). Our algorithm is more accurate than kTail for our service scenarios because we leverage a fact that the request processing starts with the receiving of the request, and hence, has the same single initial state *St0* for all the FSAs.

Algorithm 2 Combine per-path FSAs
Input: F_s, a set of per-path FSAs
Output: the core FSA F_{core} and the full FSA F_{full}
1. initialize a hashtable ***path_count*** // count how many FSAs a path is in
2. **For** each per-path FSA F **in** F_s :
3. **For** each state S **in** F :
4. state_path SP = ***transition_path***(S) // a sequence from *St0* to S in F
5. **If** SP **not in** ***path_count*** :
6. add SP to ***path_count***
7. ***path_count***(SP) = 1
8. **else**
9. ***path_count***(SP) + =1
10. **For** each state_path SP **in** ***path_count*** :
11. **If** ***path_count***(SP)= **size**(F_s) : // the path is in all per-path FSAs
12. Merge(F_{core} , SP)
13. Merge(F_{full} , SP)
14. **else**
15. ***merge***(F_{full} , SP)
16. **Return** F_{core}, F_{full}
17. Merge(F , SP):
18. Match SP against F starting from *St0*
19. **If** SP **not in** F, i.e. a branching state S_{br} is identified :
20. link the rest sequence of SP into F at S_{br}

4.2 Detection Stage

The detection process starts when a request is first received by the service platform, e.g. at the portal of a cloud platform, at the first REST API component of a request processing system, or at the Hadoop component that receives a user-initiated job request. Then the request type is identified, the corresponding aggregated FSAs for the request type are selected, and the anomaly detection is performed against the aggregated FSAs. This is an online anomaly detection while the request is being processed.

4.2.1 Identifying the request type

As the REPTrace intercepts the network messages to and from all components of the target service platform, we add platform-specific handling in the interception of the component that receives external requests from clients. The platform-specific handling parses the external requests, identifies the request type, and sends the request type to the Central Generator (see Figure 2). Then the aggregated FSAs for this identified type of request are selected for anomaly detection in the following step.

4.2.2 Detecting Anomaly of Request Execution

Figure 9 shows the overall procedure of detecting execu-

tion anomaly against the two aggregated FSAs. Three modules in the Central Generator run in parallel for detection.

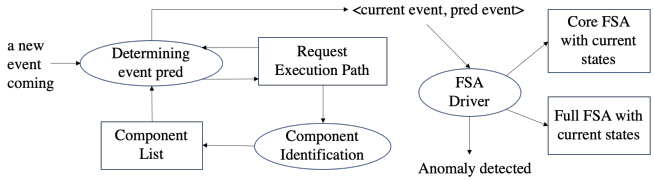


Figure 9: Detection of execution anomaly against aggregated FSAs

When a new REPTrace event is reported to the Central Generator, this "determining event pred" module identifies the pred event of the incoming trace event based on the current REP, and links the incoming event into this REP. The arbitration described in Section 3.1 is applied for multi-pred cases. The time complexity of this step is *O(n)* where n is the current number of events in this REP. The component information from the component list is also added to the incoming event. Then the event and its pred event are passed to the FSA driver. At the same time, the "component identification" module processes the updated REP and amends the component list as described in Section 3.2.

As the request may be executed by multiple components of the service platform in parallel at a time, multiple events may be produced concurrently. This means the current state of the service platform in processing the request is actually captured as a set of active FSA states in both the core FSA and the full FSA, similar to tokens in a classical Petri-Net.

As a trace event maps into an FSA edge, when a pair of *<current event, pred event>* arrives, the FSA driver identifies the source FSA state in both the FSAs, which is one of the active FSA states and has an FSA edge matching the current event (matching of *ftype* and *component_id*), and transitions the activeness of the source state to the next FSA state accordingly. Those events causing concurrency, like *pthread_create/fork/vfork* trace events, generate two transitions and one active FSA state results in two active states as one more thread/process is created.

Detecting Anomalies. In one of the following conditions, an anomaly is detected.

- The *<pred event, current event>* pair cannot trigger a transition in the full FSA, i.e., the match fails, and the source FSA state cannot be identified.
- At least one transition of the core FSA is not traversed by the execution when the request processing finishes. As events arrive frequently during the execution, when no event of the request processing arrives within a threshold period of time, it is considered the request processing finished. We select 5 seconds as the empirical threshold value in our experiments because the inter-event intervals in our experiments are typically tens to hundreds of milliseconds and 5 seconds is long enough.

5 Experimental Evaluation

We implemented a prototype of REPTrace on Linux distributions and the source code can be found on GitHub [18]. Our current implementation intercepts 28 LIBC calls via the LD_PRELOAD mechanism on Linux, including the POSIX library calls on network operations and process/thread operations as listed in Table 1, and emits 28 different types of events.

Experiment Setup. The scenarios evaluated include:

a) comparison of the tracings of the WordCount application on Hadoop (including Yarn, MapReduce and HDFS) by REPTrace and vPath, for evaluating the correctness and completeness of the REPTrace;

b) overhead analysis for the tracings of applications by REPTrace, on Hadoop, Spark, Tensorflow (for image classification using SVM) and Angel [19] (a popular platform for serving machine learning jobs);

c) REPTrace-based knowledge extraction for two service-computing software stacks, Hadoop and Angel;

d) REPTrace-based anomaly detection for four typical applications on Hadoop (i.e. four request types, as a different application is regarded as a different request type for the Hadoop job processing system): Kmeans, WordCount, Grep and TopN;

e) comparison of our anomaly detection with other leading detection approaches on a miscroservice system called Train-ticket [36], which is composed of 41 microservices to provide online train ticket booking functions.

The workloads in our experiments include WordCount, Grep, Kmeans, TopN, Teragen and Terasort on Hadoop, WordCount on Spark, Logistic Regression on Angel, SVM on Tensorflow and Ticket Booking on Train-ticket. These workloads are classical and common workloads for Hadoop, Spark, TensorFlow, Angel and Train-ticket, respectively. For each application (request type) we run 10 jobs (requests) for overhead evaluation, 20 jobs for the training stage of the anomaly detection, and 28 normal jobs, 48 fault injected jobs for the detection stage. The experiments in Section 5.1-5.4 were conducted on 2 KVM virtual machines on a Dell R730 server. Each VM has 4 cores of Intel Xeon E5-2603, 4GB main memory and Ubuntu 16.04. The experiments in Section 5.5 were conducted in a Docker container-based environment where REP-Agents and Local Generators are deployed inside containers.

5.1 Request Execution Path Analysis

We conducted experiments to compare vPath and REPTrace on tracing of the WordCount application on Hadoop with a 500M input file. We got the vPath source code from the authors of the vPath.

vPath averagely generates 28,908 trace events while REPTrace averagely generates 39,555 trace events. We investigate a subset of the events and confirm the events and their relationships captured by vPath are all captured by REPTrace. However, vPath produces 1,153 execution path fragments, while REPTrace produces 6 execution path fragments for one job execution before the handling of data dependency relationships in REPTrace is invoked. vPath produces much larger number of fragments because it only supports a constrained communication/thread pattern.

After we invoke REPTrace's handling of Hadoop job ID/RPC ID to resolve Hadoop's data dependency relationships for event linking (this is done through a Hadoop plugin we developed to extract job IDs and RPC IDs

from intercepted messages), the 6 execution path fragments are linked together to form a single complete REP for the job request. We looked into the events of several REP paths, and did not find any error in inspected relationships of the paths (via manual inspection of a number of generated REPs and relevant source code), i.e. all inspected ones reflect the real event relationships as expected and as specified in Section 2.2.

5.1.1 Case Analysis

vPath reports more than one thousand path fragments for processing of one request because Hadoop RPC and certain communication mechanisms in Hadoop do not support vPath's assumed pattern, which is described in Section 1.

Figure 10(a) shows a simplified Hadoop RPC client implementation. An RPC client calls *call()* to send an RPC request to an RPC server, which creates a new connection and sets up I/O streams for this connection. In *setupIOstreams()*, the RPC client calls *getNegotiatedProperty()* to receive necessary authentication information from the connection. Then the RPC client launches a thread to receive RPC responses and another thread to send the RPC request via this connection. The expected REP fragmemt for this scenario (starting from receiving the negotiation information), which is exactly the REP fragment generated by REPTrace, is shown in Figure 10(b). The REP fragments produced by vPath are shown in Figure 10(c). The events of the same request are identified as three path fragments by vPath. Here are two cases when vPath fail to link events correctly.

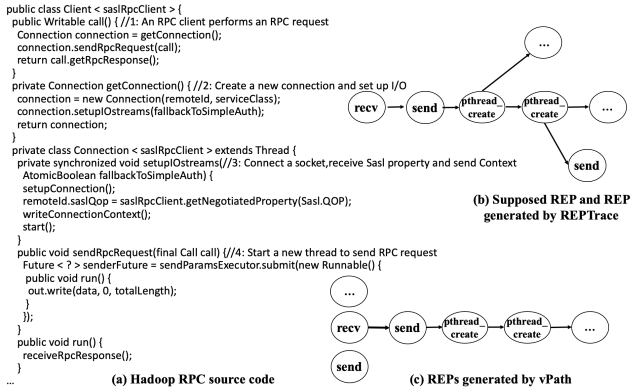


Figure 10: Hadoop RPC and its partial REP

Case 1. vPath assumes that after receiving a request X (*recv-request-X* event), the same thread will emit subordinate request Y (*send-request-Y*) to another component, block itself, receive the reply from that component, and then reply to request X (*send-reply-X*). But this assumption fails to hold in the scenario above where the *recv-request-X* event and *send-request-Y* event occur in different threads. This request-reply pattern is widely used in Hadoop, especially in Hadoop RPC which is used by Hadoop components to communicate with each other.

So vPath fails to capture the relationships between one thread's *recv* event and the *send* events of the thread's child threads. This thread pattern is handled by REPTrace propagating MSG_CTX_ID across thread/process creation, as discussed in Section 2.2 *Type 1* and *Type 2* parts.

Case 2. vPath also assumes synchronous network communication that a thread sends a request message over a TCP connection and then blocks until the corresponding reply message comes back over the same TCP connection. However, as shown in Figure 10(a), RPC client starts a thread to perform *sendRpcRequest()* and another thread to perform *receiveRpcResponse()*. The *send* and *recv* are asynchronous and are not done within the same thread.

Therefore, vPath fails to connect these two events and produces two unrelated path fragments. REPTrace does not depend on the synchronous communication pattern. Instead, REPTrace employs MSG_ID and MSG_CTX_ID cascadingly to maintain request context across components and link the asynchronous *send* and *recv* in two threads into the same REP.

5.2 Overhead Analysis

There are two types of overhead in using REPTrace:

- Latency Overhead = (average latency with REPTrace – average latency without REPTrace)/average latency without REPTrace
- Traffic Overhead = (Total of all transmitted messages' sizes with REPTrace – Total of all transmitted messages' sizes without REPTrace) / Total of all transmitted messages' sizes without REPTrace

In the experiments for the overhead analysis we first run a target application on a system, as listed in Table 2, without REPTrace deployed, and then run the same scenario with REPTrace deployed. In each experiment we collect all messages' latency values and their message sizes. Then we compute the latency overhead and traffic overhead as shown above. The experiment of each application running on each system in Table 2 with REPTrace deployed (and without REPTrace deployed) was executed 10 times.

Table 2 shows that REPTrace captures REPs for different distributed platforms with a 4.5% average latency overhead and a 1.8% average traffic overhead. A smaller average size of messages leads to a higher traffic overhead. Tensorflow's traffic is mainly for the parameter synchronization between the worker and the parameter server to update the parameters of machine learning models; it has much smaller message sizes compared to other distributed systems in the table. Consequently, Tensorflow's traffic overhead is higher.

Table 2: Results of the overhead analysis

Systems	Applications	Latency Overhead	Traffic Overhead
Hadoop	WordCount	6.0%	0.8%
	Teragen	4.6%	0.2%
	Terasort	4.0%	0.3%
Spark	WordCount	4.4%	0.3%
Angel	Logistic Regression	4.8%	0.9%
Tensorflow	SVM Classification	3.1%	8.2%
Average	-	4.5%	1.8%

5.3 Component Structure Extraction

The structure view of Hadoop's request processing automatically generated in our knowledge extraction experiments is depicted in Figure 11. RunJar8420, NN7414

(NameNode), DN29164, DN7577 (DataNode), RM7941 (ResourceManager), NM29308 and NM8117 (NodeManager) are the identified components of Hadoop.

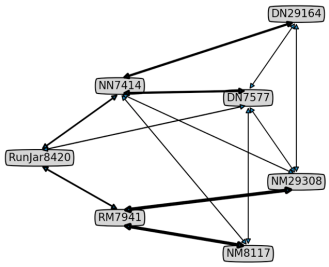


Figure 11: The structure views for Hadoop (the thickness of the links represents number of communications)

To evaluate the correctness of our knowledge extraction, we compare the structure view obtained from our experiments with the distributed system's architecture reported in the official documentation. Figure 12 is the Hadoop architecture reported in the official documentation [20]. We compare Figure 11 and Figure 12, and have the following observations:

i) Figure 11 captures all components and links in Figure 12 except for those related to ApplicationMaster and Container. It is because there are tens of container processes in the process tree, and these container processes are at the same tree depth as the ApplicationMaster processes (they are child processes of NodeManager). Recall that our component identification module ignored those processes at certain process tree depths for charting neatness if the total number of selected processes is no less than 15. If we sacrifice the charting neatness, the two types of components and relevant links are also shown in our results.

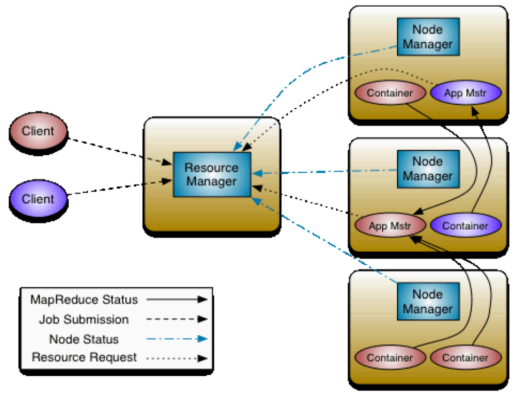


Figure 12: Official architecture of Hadoop on Yarn [20]

ii) Our approach not only identifies Yarn components (RM, NM), but also identifies HDFS components (NN, DN). We manually searched the official Hadoop documentation and did not find a holistic architecture figure that provides knowledge for both Yarn and HDFS parts of Hadoop. This observation demonstrates that our approach is able to deliver a holistic view of the distributed system while the holistic view is usually missing in official documentation (users may only find architecture of separate software there). As providing the holistic view of the system in processing of an individual request is one motivation of REPTrace, this observation justifies its motivation and verifies it achieves the goal.

5.4 Anomaly Detection

During the training stage we run 20 jobs for each of the 4 applications on Hadoop (see the experiment setup at beginning of Section 5 for details), with different input files of different sizes. The numbers of trace events vary with the input size. For a WordCount job with 500M input file, the average number of trace events in the reported REPs is 39,550 and the built full FSA has 16,297 states.

Fault injection. We use fault injection to evaluate the detection performance. We inject 6 categories of failures: Output path collision, Namenode crash, Datanode crash, Resource Manager crash, NodeManager crash and Runjar crash. For the output path collision, we started a Hadoop job request and then had another task to lock the output path of the request processing. For other categories of failures, we killed the corresponding component process after we confirmed the process is running and the processing of the request already started.

Results. For each application, we ran 28 normal jobs and 48 jobs with fault injection (8 jobs for each of the 6 failure categories). Therefore, there are 304 jobs in total, 112 without fault injection and 192 with fault injection. We use all of the 20 REPs constructed during training to train the anomaly detection; the results are summarized in Table 3.

Table 3: Anomaly detection results for four applications (FSA-20)

Application	Precision	Recall	F1-measure
Grep	0.98	0.85	0.91
Kmeans	0.92	1	0.96
TopN	0.84	1	0.91
WordCount	1	1	1
Average	0.93	0.96	0.94

We also evaluate the impacts of the number of training REPs and event consolidation on the detection performance. The anomaly detection results for four models are listed in Table 4.

- *FSA* in the table refers to an FSA model directly built from one REP for each application without pruning loop or concurrency. As only one REP path is used, there is only one FSA constructed (no core FSA or full FSA).
- *eFSA* refers to an FSA model built from one REP for each application with pruning of loop and concurrency. There is only one FSA and no core FSA or full FSA.
- *FSA-n* refers to a core FSA and a full FSA built from n training REPs (n<=20 as 20 jobs are executed in training) for each application, with loop and concurrency pruning.

Table 4: Anomaly detection results with different models

Model	Precision	Recall	F1-measure
FSA	0.034	1	0.07
eFSA	0.426	1	0.60
FSA-10	0.837	1	0.91
FSA-20	0.930	0.96	0.94

Table 4 shows that without pruning of loop and concurrency, *FSA* has very low precision, which indicates huge false positives because normal jobs' execution paths differ with each other due to iteration, concurrency, fault tolerance or built-in non-determinancy. *eFSA* improves the precision from *FSA* and reduces the false postives after pruning loop and concurrency, but the false positives

are still pretty high. *FSA-10* and *FSA-20*, by combining the core FSA and the full FSA from multiple normal REPs, alleviate the false positive issue to an acceptable level. With FSA-20, the average anomaly detection precision is 93% and the recall is 96%, which shows that the REP-Trace-based Anomaly Detection (READ) has a strong capability to detect anomalies.

5.5 Detection Result Comparison

Deep learning-based approaches are the leading approaches that leverage logs and monitoring data to build the detection models. Deeplog [37] builds a prediction model with a Long Short-Term Memory (LSTM) neural network from log data, and MRD [38] builds a Gated Recurrent Unit (GRU) neural network from monitoring data. We applied READ, Deeplog and MRD on Train-ticket with the Ticket Booking service request to compare their abilities to detect anomalies. During the training stage we emitted 1,000 ticket-searching service requests, which involve 12 microservices' interactions, and collected the request exection paths, logs and monitoring data (including CPU/Memory/Disk/Network usage of each microservice) for each request. So our training dataset has 1000 normally executed requests.

We inject software faults with SSFI [39] which is able to inject 12 different types of software faults for distributed systems. For each service request of ticket searching, we randomly inject one software fault of the 12 types into Train-ticket and then record the details of the injected faults. After collecting the logs, trace events and monitoring data during the processing of each request, the whole system is restarted for the next service request. We conducted thousands of fault injections and obtained 193 cases with anomaly manifestation (e.g. execution hangs, early job termination without completion, etc.). Then we ran another 193 ticket-searching requests without fault injection, and put these 193 cases into our test dataset. So our test dataset has 193 requests with different types of failures/anomalies and 193 normally executed requests.

Table 5: Anomaly detection results with different methods

Methods	Precision	Recall	F1-measure
READ	0.93	0.88	0.90
Deeplog	0.95	0.76	0.84
MRD	0.67	0.69	0.70

Table 5 summarizes the detection results of the three mothods on the test dataset from Train-ticket. READ outperforms Deeplog and MRD in terms of recall and F1-measure. Deeplog can accurately learn the transitions of log messages, which accounts for the high precision of detecting anomalies. But Deeplog treats each microservice as an independent component and detects anomalies for each microservice separatedly, and fails to identify those anomalies that do not manifest as unusual log transitions within a single microservice (0.76 recall in Table 5): unusual interactions among different microservices are not detected by DeepLog, nor are incorrect executions that result in incomplete log sequences. READ learns the global system state transitions based on the holistic view of the request processing executions, including both the interactions among different microservices and those within each single microservice. MRD does not perform well because lots of anomalies do not manifest as abnormal resource consumption patterns, which is the basis for MRD.

6 Related Work

6.1 End-to-End Tracing Systems

There exist various implementations of end-to-end tracing in both academia [7][8][10][11][12][14][15][21] and industry [9][13][22]. Stardust [7] is an end-to-end tracing system designed for specific distributed storage system, which requires manual modification of the storage system to generate the request execution path. Xtrace [8] extends common network protocols to support request identifier propagation by means of special hardware and application support of the protocol extension. Google's Dapper [9] instruments Google's internal homogeneous control flow/RPC libraries to do the tracing. It requires all traced components use the instrumented internal libraries. Pinpoint [10] instruments J2EE libraries to accurately correlate network communication events by adding request identifier in HTTP headers. HDFS HTrace [13] is a Dapper-like end-to-end tracing framework, which has been integrated with HDFS and HBase.

Stardust requires manual modification of the distributed storage system. Pinpoint, Dapper, Pivot [11], Magpie [12], and Facebook's Canopy [14] deal with specific middleware or vendor custom libraries only. HTrace requires instrumentation to the system to enable tracing. Certain hardware and application support is demanded for Xtrace to work.

A project close to our work is vPath [15]. This project made a thrust to provide a generic solution of tracing distributed systems/applications by monitoring thread and network activities. But vPath only deals with TCP-based system/applications and assumes certain communication styles and thread patterns. As explained in Section 1 and Section 5.1, vPath does not address our problem.

There are also studies that apply statistical technologies to logs of the distributed components for correlating log events. They apply statistical analysis [23][24], code analysis [25], time series analysis [26], or timestamps [27][28] to speculate the request execution path. Because many events are not exposed by the logs, these technologies cannot capture the accurate complete execution path of fine-grained events for individual request and just try to probabilistically matching logs events with statistical inference or identify certain statistical patterns.

6.2 Anomaly Detection of Distributed Systems

Deeplog [37] builds a prediction model with Long Short-Term Memory (LSTM) neural network to detect anomalies by checking whether the next log event is in the few predicted log event candidates. Fu [29] leverages log data and assumes that there is a thread ID or request ID for every log event to classify all the logs into different log sequence according to their ID information and timestamp. An FSA is built for each component of a distributed system based on the log sequences to detect anomalies. Due to its sequence model, it fails to capture the concurrency in request processing and is limited to a single component without correlating the log events from other components or hosts. KeWei [32] proposed an FSA-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2022.3149949

based detection approach. This approach parses log events, uses <*component, log point*> tuples as the finite states, and uses temporal order as the transitions for states to build an FSA for all possible service requests. But temporal order is not accurate to capture the transitions between these states for building an accurate FSA. Jiang [42] focused on the correlation between different metrics at the same time point and used an autoregressive model to learn the correlation to detect anomalies. MRD [38] leverages a Gated Recurrent Unit (GRU) neural network to train a model to detect anomalies by checking whether the predicted resource consumption sequence of a service request is similar to the real resource consumption sequence. Malhotra [30] adopted a similar method with MRD but employed Long Short-Term Memory (LSTM) networks to predict the metrics at a time point based on the metric values in the previous time windows. If the real metric values deviate a lot from the predicted ones, an anomaly is detected. Sambasivan et al. [31] diagnose anomalies by comparing execution paths generated by Stardust. It identifies parallel substructures of request execution but fails to find the iteration substructures.

6.3 Extracting Knowledge on System Behavior

Existing mechanisms of extracting knowledge on system behavior fall in the following categories: a) static analysis which requires source code availability; b) tools focusing on understanding/visualization of sequential programs or sequential fragments of an application [33][34]; c) statistic mechanisms that compute certain statistic metric values or generate statistic summary/correlations from a large number of data traces, for example, MagPie [12]; d) impact analysis of a specific component or part of the distributed system, i.e. how this component's failure affects other components' execution [35]; and e) software behavior models inferred from logs or method invocation traces, such as GK-tail [45], Synoptic [44] and SpecMiner [43]. Methods of the last category model software executions from logs or method invocation traces as an FSA and then applies kTail algorithm [41] or its extensions to extract a final FSA from multiple FSAs to represent the software behavior. But they mainly deal with sequential executions within threads, and do not address the service computing scenarios where the same thread handles multiple different requests concurrently, a request's processing traverses threads/processes across multiple nodes, and different requests' processings should be separated.

The focus of our work is on obtaining certain knowledge and understanding of the distributed services' behavior in processing individual service request, in a holistic view. Specifically, we intend to capture the overall component structure/architecture of a distributed system's request processing, reveal the complete in-depth interactions of the components (and their subcomponents/processes/threads) along the processing of an individual service request, and perform feature studies on the distributed systems' certain reliability mechanisms. Moreover, our methodology extracts the knowledge of distributed systems without requiring source code, documentation, or other prior knowledge.

7 Conclusions

In this paper, we propose REPTrace, a generic end-to-end tracing methodology, which automatically generates the complete execution path of service requests for a variety of computing platforms in a transparent way. We analyzed possible request execution scenarios during the platforms' request processing, and introduced how REPTrace captures the accurate event relationships in these scenarios. We conducted experiments to compare REPTrace with vPath. Our results show that REPTrace has much better performance in terms of generality and accuracy because REPTrace is not limited to certain network communications styles or thread patterns as vPath is. In our experiments REPTrace has a low execution latency overhead (4.5%) and low network traffic overhead (1.8%).

To demonstrate the effectiveness of REPTrace in understanding and diagnosis of the service behavior, we also devise an anomaly detection mechanism based on REPTrace to detect anomalies of request execution. The experiments on Hadoop show that anomalies are detected with 93% precison and 96% recall by leveraging the REPs. The experiments also show that with the collected REP data we can construct the structure of the whole stack of service computing accurately. Moreover, the comparison of our anomaly detection with other leading anomaly detection approaches shows that our REPTrace-based anomaly detection outperforms them because REPTrace provides the global holistic view of request processing which does not exist in the other approaches.

Acknowledgment

This work has been supported by Key-Area Research and Development Program of Guangdong Province (No. 2020B010164003). Long Wang and Ying Li are the corresponding authors.

References

[1] https://aws.amazon.com/machine-learning/
[2] IBM Watson Health, https://www.ibm.com/watson-health/healthcare-consulting
[3] https://cloud.google.com/products
[4] OpenStack Releases, https://releases.openstack.org/
[5] Cornelissen B, et al., A systematic survey of program comprehension through dynamic analysis. IEEE Trans. onSoftware Engineering,35(5).
[6] Salman A. Baset, Chunqiang Tang, Byung Chul Tak, Long Wang, "Dissecting Open Source Cloud Evolution: An OpenStack Case Study," 5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud), San Jose, CA, USA, 2013.
[7] R. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, G.R. Ganger, "Stardust: Tracking Activity in a Distributed Storage System," ACM SIGMETRICS Performance Evaluation Review. ACM, 2006, 34(1): 3-14.
[8] R. Fonseca, G. Porter, R.H. Katz, I. Stoica, "X-trace: a Pervasive Network Tracing Framework," Proceedings of the 4th USENIX conference on Networked systems design & im- plementation. USENIX Association, 2007: 20-20.
[9] G.H. Sigelman, L.A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, C. Shanbhag, "Dapper, a Large-scale Distributed System Tracing Infrastructure," Technical report, Google, Inc, 2010.
[10] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services," Dependable Systems and Networks, 2002. DSN 2002. Proceedings. In- ternational Conference on. IEEE, 2002: 595-604.
[11] J. Mace, R. Roelke, R. Fonseca, "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems," Proceedings of the 25th

based detection approach. This approach parses log events, uses <*component, log point*> tuples as the finite states, and uses temporal order as the transitions for states to build an FSA for all possible service requests. But temporal order is not accurate to capture the transitions between these states for building an accurate FSA. Jiang [42] focused on the correlation between different metrics at the same time point and used an autoregressive model to learn the correlation to detect anomalies. MRD [38] leverages a Gated Recurrent Unit (GRU) neural network to train a model to detect anomalies by checking whether the predicted resource consumption sequence of a service request is similar to the real resource consumption sequence. Malhotra [30] adopted a similar method with MRD but employed Long Short-Term Memory (LSTM) networks to predict the metrics at a time point based on the metric values in the previous time windows. If the real metric values deviate a lot from the predicted ones, an anomaly is detected. Sambasivan et al. [31] diagnose anomalies by comparing execution paths generated by Stardust. It identifies parallel substructures of request execution but fails to find the iteration substructures.

6.3 Extracting Knowledge on System Behavior

Existing mechanisms of extracting knowledge on system behavior fall in the following categories: a) static analysis which requires source code availability; b) tools focusing on understanding/visualization of sequential programs or sequential fragments of an application [33][34]; c) statistic mechanisms that compute certain statistic metric values or generate statistic summary/correlations from a large number of data traces, for example, MagPie [12]; d) impact analysis of a specific component or part of the distributed system, i.e. how this component's failure affects other components' execution [35]; and e) software behavior models inferred from logs or method invocation traces, such as GK-tail [45], Synoptic [44] and SpecMiner [43]. Methods of the last category model software executions from logs or method invocation traces as an FSA and then applies kTail algorithm [41] or its extensions to extract a final FSA from multiple FSAs to represent the software behavior. But they mainly deal with sequential executions within threads, and do not address the service computing scenarios where the same thread handles multiple different requests concurrently, a request's processing traverses threads/processes across multiple nodes, and different requests' processings should be separated.

The focus of our work is on obtaining certain knowledge and understanding of the distributed services' behavior in processing individual service request, in a holistic view. Specifically, we intend to capture the overall component structure/architecture of a distributed system's request processing, reveal the complete in-depth interactions of the components (and their subcomponents/processes/threads) along the processing of an individual service request, and perform feature studies on the distributed systems' certain reliability mechanisms. Moreover, our methodology extracts the knowledge of distributed systems without requiring source code, documentation, or other prior knowledge.

7 Conclusions

In this paper, we propose REPTrace, a generic end-to-end tracing methodology, which automatically generates the complete execution path of service requests for a variety of computing platforms in a transparent way. We analyzed possible request execution scenarios during the platforms' request processing, and introduced how REPTrace captures the accurate event relationships in these scenarios. We conducted experiments to compare REPTrace with vPath. Our results show that REPTrace has much better performance in terms of generality and accuracy because REPTrace is not limited to certain network communications styles or thread patterns as vPath is. In our experiments REPTrace has a low execution latency overhead (4.5%) and low network traffic overhead (1.8%).

To demonstrate the effectiveness of REPTrace in understanding and diagnosis of the service behavior, we also devise an anomaly detection mechanism based on REPTrace to detect anomalies of request execution. The experiments on Hadoop show that anomalies are detected with 93% precison and 96% recall by leveraging the REPs. The experiments also show that with the collected REP data we can construct the structure of the whole stack of service computing accurately. Moreover, the comparison of our anomaly detection with other leading anomaly detection approaches shows that our REPTrace-based anomaly detection outperforms them because REPTrace provides the global holistic view of request processing which does not exist in the other approaches.

Acknowledgment

This work has been partially supported by Key-Area Research and Development Program of Guangdong Province (No. 2020B010164003) and the NSFC Project of China under Grant 62132009. Long Wang and Ying Li are the corresponding authors.

References

[1] https://aws.amazon.com/machine-learning/

[2] IBM Watson Health, https://www.ibm.com/watson-health/healthcare-consulting

[3] https://cloud.google.com/products

[4] OpenStack Releases, https://releases.openstack.org/

[5] Cornelissen B, et al., A systematic survey of program comprehension through dynamic analysis. IEEE Trans. onSoftware Engineering,35(5).

[6] Salman A. Baset, Chunqiang Tang, Byung Chul Tak, Long Wang, "Dissecting Open Source Cloud Evolution: An OpenStack Case Study," 5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud), San Jose, CA, USA, 2013.

[7] R. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, G.R. Ganger, "Stardust: Tracking Activity in a Distributed Storage System," ACM SIGMETRICS Performance Evaluation Review. ACM, 2006, 34(1): 3-14.

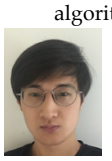
[8] R. Fonseca, G. Porter, R.H. Katz, I. Stoica, "X-trace: a Pervasive Network Tracing Framework," Proceedings of the 4th USENIX conference on Networked systems design & im- plementation. USENIX Association, 2007: 20-20.

[9] G.H. Sigelman, L.A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, C. Shanbhag, "Dapper, a Large-scale Distributed System Tracing Infrastructure," Technical report, Google, Inc, 2010.

[10] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services," Dependable Systems and Networks, 2002. DSN 2002. Proceedings. In- ternational Conference on. IEEE, 2002: 595-604.

[11] J. Mace, R. Roelke, R. Fonseca, "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems," Proceedings of the 25th

Symposium on Operating Systems Principles. ACM, 2015: 378-393.
[12] P. Barham, R. Isaacs, R. Mortier, D. Narayanan, "Magpie: Online Modelling and Performance-aware Systems," HotOS. 2003: 85-90.
[13] HTrace, http://htrace.incubator.apache.org/
[14] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neal, K.W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, Y.J. Song, "Canopy: an End-to-end Performance Tracing and Analysis System," ACM Symposium on Operating Systems Principles, 2017
[15] B.C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, R.N. Chang, "vPath: Precise Discovery of Request Processing Paths from Black-box Observations of Thread and Network Activities," USENIX Annual technical conference. 2009
[16] Y. Yang, L. Wang, J. Gu, Y. Li. "Transparently Capturing Execution Path of Service/Job Request Processing," International Conference on Service-Oriented Computing (ICSOC) 2018. Lecture Notes in Computer Science, vol 11236.
[17] J. Gu, L. Wang, Y. Yang, Y. Li, "KEREP: Experience in Extracting Knowledge on Distributed System Behavior through Request Execution Path," 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 2018: 30-35.
[18] REPTrace, https://github.com/alexvanc/REPTrace
[19] Angel Project, https://github.com/Tencent/angel
[20] https://hadoop.apache.org/docs/r2.6.5/
[21] P. Reynolds, C.E. Killian, J.L. Wiener, J.C. Mogul, M.A. Shah, A. Vahdat, "Pip: Detecting the Unexpected in Distributed Systems," NSDI. 2006, 6: 9-9.
[22] Zipkin, https://zipkin.io/
[23] M. Chow, D. Meisner, J. Flinn, D. Peek, T.F. Wenisch, "The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services," OSDI. 2014: 217-231.
[24] A. Anandkumar, C. Bisdikian, D. Agrawal, "Tracking in a Spaghetti Bowl: Monitoring Transactions Using Footprints," In SIGMETRICS '08: Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pages 133-144, New York, NY, USA, 2008. ACM.
[25] X, Zhao, Y. Zhang, D, Lion, M.F. Ullah, Y. Luo, D. Yuan, M. Stumm, "lprof: a Non-intrusive Request Flow Profiler for Distributed Systems," OSDI. 2014, 14: 629-644.
[26] T. Wang, C. Perng, T. Tao, C. Tang, E. So, C. Zhang, R. Chang, L. Liu, "A Temporal Data-mining Approach for Discovering End-to-end Transaction Flow," Web Services, 2008. ICWS'08. IEEE International Conference on. IEEE, 2008: 37-44.
[27] K. Kc, X. Gu, "ELT: Efficient Log-based Troubleshooting System for Cloud Computing Infrastructures," Reliable Distributed Sys-tems (SRDS), 2011 30th IEEE Symposium on. IEEE, 2011: 11-20.
[28] B.C. Tak, S. Tao, L. Yang, C. Zhu, Y. Ruan, "LOGAN: Problem Diagnosis in the Cloud Using Log-based Reference Models," Cloud Engineering (IC2E), 2016 IEEE International Conference on. IEEE, 2016: 62-67.
[29] Q. Fu, J.G. Lou, Y. Wang, J. Li, "Execution Anomaly Detection in Distributed Systems Through Unstructured Log Analysis," Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on. IEEE, 2009: 149-158.
[30] P. Malhotra, L. Vig, G. Shroff, and P. Agarwal, "Long short term memory networks for anomaly detection in time series," in Proceedings. Presses universitaires de Louvain, 2015, p. 89.
[31] Sambasivan, Raja R., 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, vol. 5, pp. 1-1. 2011.
[32] Kewei S, Jie Q, Ying L, Ying C, Weixing J, "A State Machine Approach for Problem Detection in Large-scale Distributed System." NOMS 2008-2008 IEEE Network Operations and Management Symposium. IEEE, 2008.
[33] Pacione M J, Roper M, Wood M. A novel software visualisation model to support software comprehension[C]//Reverse Engineering, 2004. Proceedings. 11th Working Conference on. IEEE, 2004: 70-79.
[34] Cornelissen B, Zaidman A, van Deursen A. A controlled experiment for program comprehension through trace visualization[J]. IEEE Transactions on Software Engineering, 2011, 37(3).
[35] Cai H, Thain D. Distia: A cost-effective dynamic impact analysis for distributed programs, Proc. of 31st IEEE/ACM International Conference on Automated Software Engineering. ACM, 2016.
[36] https://github.com/FudanSELab/train-ticket
[37] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communica- tions Security. ACM, 2017, pp. 1285–1298.
[38] A. Y, "Using gru to predict the resource consumption sequence of service requests," https://github.com/alexvanc/gru resource prediction.
[39] Yang, Yong, et al. "How far have we come in detecting anomalies in distributed systems? an empirical study with a statement-level fault injection method." 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2020.
[40] https://sourceforge.net/projects/libuuid/
[41] Biermann, Alan W., and Jerome A. Feldman. "On the synthesis of finite-state machines from samples of their behavior." IEEE transactions on Computers 100.6 (1972): 592-597.
[42] G. Jiang, H. Chen, and K. Yoshihira, "Modeling and tracking of transaction flow dynamics for fault detection in complex systems," IEEE Transactions on Dependable and Secure Computing, vol. 3, no. 4, pp. 312–326, 2006.
[43] Kabir, Muhammad Ashad, et al. "SpecMiner: Heuristic-based mining of service behavioral models from interaction traces." Future Generation Computer Systems 117 (2021): 59-71.
[44] Beschastnikh, Ivan, et al. "Synoptic: Studying logged behavior with inferred models." Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 2011.
[45] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. "Automatic generation of software behavioral models." Proceedings of the 30th international conference on Software engineering. 2008.
[46] T H Cormen, C E Leiserson, R L Rivest, et al. Introduction to algorithms. MIT press, 2009.



Yong Yang received the PhD degree from Peking University in 2020. His research interests include cloud computing, system dependability, anomaly detection and distributed systems.

Long Wang is currently an associate professor of Tsinghua University. Prior to that, he was a senior research staff member of IBM T. J. Watson Research Center leading the resiliency and security department of IBM Watson Health Cloud. He got Ph.D. in Electrical and Computer Engineering from University of Illinois at Urbana Champaign in 2010, and then joined IBM T. J. Watson Research Center. He has published 40+ papers in international technical journals and conferences and has filed more than 25 patent disclosures. His research interests include dependable and secure systems, distributed systems, cloud computing, system modeling, measurement and assessment, big data analytics, and machine learning. He is a senior member of the IEEE.

Jing Gu received the BE degrees in software engineering from Central South University in 2016. She got B.S in the Department of Software Engineering, Peking University in 2019. She is working as a software engineer with Alibaba, Hangzhou, China. Her research interests include cloud computing, distributed system, and big data.

Ying Li received the PhD degree in computer science and engineering from Northwestern Polytechnical University (NWPU), China, in 2001. She is currently a professor in the school of Software and Microelectronics, Peking University, China. Before joining PKU in 2012, she worked as a STSM and senior manager leading the department of distributed computing in IBM China Research Center. She was rewarded with "IBM Global Research Accomplishment Award" twice and "CIO Leadership Award". Her research interests include distributed computing and dependability engineering. She filed 30+ US/CN granted patents and was awarded as "IBM Master Inventor". She published 60+ papers in international journals and conferences and served as PC member of several international conferences and reviewer of international journals. She is a member of the IEEE.