Precise, Scalable, and Online Request Tracing for Multitier Services of Black Boxes

Bo Sang, Jianfeng Zhan, Gang Lu, Haining Wang, *Senior Member, IEEE*, Dongyan Xu, Lei Wang, Zhihong Zhang, and Zhen Jia

Abstract—As more and more multitier services are developed from commercial off-the-shelf components or heterogeneous middleware without source code available, both developers and administrators need a request tracing tool to 1) exactly know how a user request of interest travels through services of black boxes and 2) obtain macrolevel user request behaviors of services without manually analyzing massive logs. This need is further exacerbated by IT system "agility," which mandates the tracing tool to provide online performance data since offline approaches cannot reflect system changes in real time. Moreover, considering the large scale of deployed services, a pragmatic tracing approach should be scalable in terms of the cost in collecting and analyzing logs. In this paper, we introduce a precise, scalable, and online request tracing tool for multitier services of black boxes. Our contributions are threefold. First, we propose a precise request tracing algorithm for multitier services of black boxes, which only uses application-independent knowledge. Second, we present a microlevel abstraction, *component activity graph*, to represent causal paths of each request. On the basis of this abstraction, we use *dominated causal path patterns* to represent repeatedly executed causal paths that account for significant fractions, and we further present a derived performance metric of causal path patterns, *latency percentages of components*, to enable debugging performance-in-the-large. Third, we develop two mechanisms, *tracing on demand* and *sampling*, to significantly increase the system scalability. We implement a prototype of the proposed system, called *PreciseTracer*, and release it as open source code. In comparison with WAP5—a black-box tracing approach, PreciseTracer achieves higher tracing accuracy and faster response time. Our experimental results also show that PreciseTracer has low overhead, and still achieves high tracing accuracy even if an aggressive sampling policy is adopted, indicating that PreciseTracer is a promising tracing tool for large-scale production systems.

Index Terms—Multitier service, black boxes, precise request tracing, micro- and macrolevel abstractions, online analysis, performance debugging, scalability.

1 Introduction

More and more multitier services or cloud applications [36] are deployed on data centers. In order to pinpoint performance bottlenecks, shoot misconfigurations, or even accurately tune power-saving management policies [32], both developers and administrators need a tracing tool to 1) know how exactly a user request or job of interest travels through services or job execution frameworks, e.g., a MapReduce runtime system [36] if necessary, and 2) obtain macrolevel user behaviors without analyzing massive logs. The need for online tracing is further exacerbated by IT system agility [22]. For example, when emergency events happen like earthquakes or hurricanes, requests for the related stories at popular news or weather websites will result in heavy loads that could be an order of magnitude greater than those loads in a normal condition [30]. To deal with fluctuated workloads, resources are often dynamically provisioned and service instances are adjusted accordingly. In this context, *online* performance information is imperative, which only can be obtained with an online tracing tool because an offline one cannot reflect system changes in real time. Moreover, understanding online behavior is also important under normal operating conditions for online scheduling and dispatching decisions [7] for either multitier web services [30] or high throughput computing (HTC) or many task computing (MTC) workloads in consolidated cloud computing systems [37], [38].

This paper focuses on online request tracing of multitiers services of *black boxes*, since more and more multitier services are developed from commercial off-the-shelf components or heterogeneous middleware without source code available. Different from profiling [35] that is measurement of a statistical summary of the behaviors of a system, tracing is a measurement of a stream of events of a system [39]. In this paper, when we refer to *precise request tracing*, it implies the accurate tracking of how a user request of interest travels through services.

Precise request tracing for multitier services of black boxes is challenging in many aspects. First of all, we cannot access source code for multitier services of black boxes, and

- B. Sang and D. Xu are with the Department of Computer Science, Purdue University, Lawson Computer Science Build, 305 N. University St., West Lafayette, IN 47907. E-mail: samplise@gmail.com, dxu@cs.purdue.edu.
- J. Zhan, G. Lu, L. Wang, Z. Zhang, and Z. Jia are with the Institute of Computing Technology, Chinese Academy of Sciences, No. 6 Kexueyuan South Road Zhongguancun, Haidian District, Beijing 100190, China. E-mail: {jfzhan, lugang, wl, lugang}@ncic.ac.cn.
- H. Wang is with the Department of Computer Science, College of William and Mary, PO Box 8795, Williamsburg, VA 23187. E-mail: hnw@cs.wm.edu.

Manuscript received 24 Aug. 2010; revised 31 Aug. 2011; accepted 18 Sept. 2011; published online 19 Oct. 2011.
Recommended for acceptance by J. Weissman.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2010-08-0499.
Digital Object Identifier no. 10.1109/TPDS.2011.257.

thus it is difficult to understand the contexts of requests or even network protocols [5]. Second, a precise request tracing tool is needed to exactly track a user request of interest if necessary, with the focus on *performance-in-the-large* [4]. However, services are often deployed within a large-scale data center, and a precise request tracing system will inevitably produce massive logs, and hence a pragmatic tracing tool should be scalable with respect to log collection and analysis [27]. Moreover, macrolevel abstractions are required to facilitate debugging performance-in-the-large. Finally, those tools should not degrade the performance of multitier services.

The most straightforward and accurate way [1], [10], [11], [12], [15], [27] of correlating message streams is to leverage application-specific knowledge and explicitly declare causal relationships among events of different components. Its drawback is that users must obtain and modify source code of target applications or middleware, or it might even require that users have in-depth knowledge of target applications or instrumented middleware. Thus, this approach cannot be used for services of black boxes. Without the knowledge of source code, several previous approaches [3], [4], [7] either use imprecision of probabilistic correlation methods to infer the average response time of components, or rely upon the knowledge of protocols to isolate events and requests for precise request tracing [5]. A precise but unscalable request tracing tool, called *vPath* [22], is proposed for services of black boxes. Because of its limitation in the implementation, the tracing mechanism of vPath cannot be enabled or disabled on demand without interrupting services. Thus, it has to continuously collect and analyze logs, resulting in an unacceptably high cost. Moreover, state-of-the-art precise request tracing approaches of black boxes [5], [22] fail to propose abstractions for representing macrolevel user request behaviors, and instead depend on users' manual interpretations of massive logs in debugging performance-in-the-large. Besides, these [5], [20], [22] are offline schemes.

In this paper, we present a precise and scalable request tracing tool for online analysis of multitier services of black boxes. Our tool collects activity logs of multitier services through the kernel instrumentation, which can be enabled or disabled on demand. Through tolerating log losses, our system supports *sampling* or *tracing on demand*, which significantly decreases the collected and analyzed logs and improves the system scalability. After reconstructing those activity logs into *causal paths*, each of which is a sequence of activities with causal relations caused by an individual request, we classify those causal paths into different *causal patterns*, which represent repeatedly executed causal paths. Then, we present a macrolevel abstraction, *dominated causal path patterns*, to represent causal path patterns that account for significant fractions in terms of the numbers of causal paths. On the basis of this, we propose a derived performance metric of major causal path patterns, *latency percentages of components*, to enable debugging performance-in-the-large.

We implement a prototype of the proposed system, called *PreciseTracer*, and release it as open source code.[1] We perform extensive experiments on 3-tier platforms. Our experimental results show that: 1) with respect to WAP5—a black-box tracing approach [3], PreciseTracer achieves higher tracing accuracy and faster response time; 2) PreciseTracer has low overhead, and still achieves high tracing accuracy even if an aggressive sampling policy is adopted; and 3) our derived performance metric of causal path patterns, latency percentages of components, enables debugging performance-in-the-large.

Summarily, we make the following contributions in this paper:

1. We design a precise tracing algorithm to deduce causal paths of requests from interaction activities of components of black boxes. Our algorithm only uses application-independent knowledge.
2. We present two abstractions, component activity graph (CAG) and dominated causal path pattern, to represent individual causal paths for each request and macrolevel user request behaviors, respectively. Based on these abstractions, we propose a derived performance metric, latency percentages of components, to enable debugging performance-in-the-large.
3. We present two mechanisms, tracing on demand and sampling, to improve the system scalability.

The remainder of the paper is organized as follows. Section 2 formulates the problem. Section 3 describes the design of PreciseTracer. Section 4 details the implementation of PreciseTracer. Section 5 draws a conclusion. Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.257, describes the pseudocode of PreciseTracer. Appendix B, available in the online supplemental material, evaluates the performance of PreciseTracer with respect to WAP5. Appendix C, available in the online supplemental material, summarizes related work.

2 System Model and Problem Statement

2.1 System Model

Our target environments are data centers deployed with multitier services. There are two types of nodes in these environments: *service nodes* and *analysis nodes*. Service nodes are the ones, on which multitier services are deployed, while most the components of the tracing tool are deployed on analysis nodes.

The application assumptions are as follows:

- We treat each component in a multitier service as a black box, since we cannot obtain the application or middleware source code, neither deploy the instrumented middleware, nor have the knowledge of high-level protocols used by services, like HTTP.
- We presume that a single execution entity (a process or a kernel thread) of each component can only serve one request in a certain period. For serving each individual request, execution entities of the components cooperate through sending or receiving messages via a reliable communication protocol, like TCP. An individual request is tracked by monitoring a series of activities, which have causal relations for tracing requests.

1. The source code can be downloaded from http://prof.ncic.ac.cn/DCBenchmarks/.

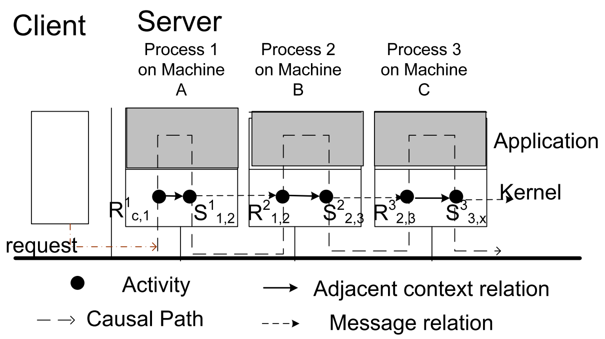


Fig. 1. Activities with causal relations in the kernel.

Though not all multitier services fall in the scope of our target applications, fortunately many popular services satisfy our assumed scenarios. For example, our method can be used to analyze concurrent servers following nine design patterns introduced in [6, Chapter 27], including iteration, concurrent process, concurrent thread, preforking, and prethreading models.

2.2 Problem Statement

As shown in Fig. 1, a request causes a series of *interaction activities* in the operating system kernel, e.g., sending or receiving messages. Those activities occur under specific contexts (*processes* or *kernel threads*) of different components. We record an activity of sending a message as $S^i_{i,j}$, which indicates a process i sends a message to a process j. We record an activity of receiving a message as $R^j_{i,j}$, which indicates a process j receives a message from a process i.

When an individual request is serviced, a series of activities having causal relations or happened-before relationships as defined by Lamport [8] constitute *a causal path*, e.g., in Fig. 1, the activity sequence $\{R^1_{c,1}, S^1_{1,2}, R^2_{1,2}, S^2_{2,3}, R^3_{2,3}, S^3_{3,x}\}$ constitutes a causal path. For each individual request, there is a causal path.

Given the scenario above, the problems we tackle in this paper can be described as follows: with the logs of communication activities, how can we obtain individual causal paths that precisely correlate activities of components for each request? How to obtain dominated causal path patterns, which represent repeatedly executed causal paths, and their online or offline performance data?

The objective of this work is that with the data provided by PreciseTracer, developers and administrators can accurately track how a user request of interest travels through services, debug performance problems of a multitier service, and provide online performance data of services for the feedback controller in the runtime power management system [32].

3 PreciseTracer Design

In this section, we first present the architecture of PreciseTracer and its abstractions. Then, we detail the tracing algorithm and the mechanisms for improving the system scalability.

3.1 PreciseTracer Architecture

PreciseTracer is flexible, and administrators can configure it according to their requirements. PreciseTracer can work under either an offline mode or an online mode. Under an offline mode, PreciseTracer only analyzes logs after collecting them for a relatively long period of time. Under an online mode, PreciseTracer collects, and analyzes logs in a simultaneous manner, providing administrators with real-time performance information.

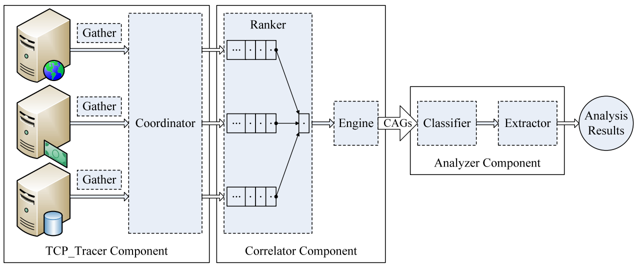


Fig. 2. Architecture of PreciseTracer.

Fig. 2 shows the architecture of PreciseTracer, which consists of three major components: *TCP_Tracer*, *Correlator*, and *Analyzer*. TCP_Tracer records interaction activities of interest for the components of the target applications, and output those logs to Correlator. Correlator is responsible for correlating those activity logs of different components into causal paths. Finally, based on the causal paths produced by Correlator, Analyzer extracts useful information, and reports analysis results.

TCP_Tracer includes two modules: *Gather* and *Coordinator*. Deployed on each service node, Gather is responsible for collecting logs of services deployed on the same node. Coordinator, which is deployed on an analysis node, is in charge of controlling Gather on each service node. Coordinator configures and synchronies Gather on each service node to send logs to Correlator.

Gather independently observes the interaction activities of the components of black boxes on each node. Concentrating on the major activities, Gather only cares about when serving a request starts, finishes, and when components receive or send messages *within the confine of a data center*. Of course, we can observe more activities if the overhead is acceptable. In this paper, our observed activity types include: *BEGIN*, *END*, *SEND*, and *RECEIVE*. SEND and RECEIVE activities are the ones of sending and receiving messages. A BEGIN activity marks the start of serving a new request, while an END activity marks the end of serving a request. For each activity, Gather records five attributes: *(activity type, time stamp, context identifier, message identifier)*, and *message size*. For each activity, we use 4-tuple *(hostname, program name, process ID, thread ID)* to describe its context identifier, and use 5-tuple *(IP of sender, port of sender, IP of receiver, port of receiver, message size)* to describe its message identifier.

Correlator includes two major modules: *Ranker* and *Engine*. Ranker is responsible for choosing candidate activities for composing causal paths. Engine constructs causal paths from the outputs of Ranker, and then reports causal paths.

Analyzer includes two major modules: *Classifier* and *Extractor*. Classifier is responsible for classifying causal paths into different patterns, while Extractor provides analysis results of causal path patterns.

3.2 Abstractions

Formally, we propose *a directed acyclic graph* $G(V, E)$ to represent causal paths, where vertices V are activities of components and edges E represent causal relations between activities. We define this abstraction as *component activity graph*. For each individual request, a corresponding CAG represents all activities with causal relations in the life cycle of serving the request.

CAGs include two types of relations: *an adjacent context relation* and *a message relation*. We formally define the two relations based on the happened-before relation [8], which is denoted as $\rightarrow$, as follows:

An adjacent context relation. Caused by the same request r, x and y are activities observed in the same context c (a process or a kernel thread), and $x \rightarrow y$ holds true. If no activity z, which satisfies the relations $x \rightarrow z$ and $z \rightarrow y$, is observed in the same context, then an adjacent context relation exists between x and y, denoted as $x \rightarrow_c y$. So, the adjacent context relation $x \rightarrow_c y$ means that x has happened right before y in the same execution entity.

A message relation. For serving a request r, if x is a *SEND* activity, which sends a message m, and y is a *RECEIVE* activity, which receives the same message m, then a message relation exists between x and y, denoted as $x \rightarrow_m y$. So, the message relation $x \rightarrow_m y$ means that x, which sends a message, has happened right before y, which receives a message, in two different execution entities.

If there is an edge from activity x to activity y in a CAG, of which $x \rightarrow_c y$ or $x \rightarrow_m y$ holds true, then x is the parent of y.

In a CAG, each activity vertex must satisfy the following property: each activity vertex has no more than two parents, and only a RECEIVE activity vertex could have two parents, with which one parent has an adjacent context relation and the other has a message relation. This has twofold reasons: first, an adjacent context relation is used to describe activities in the same process or thread caused by the same request, so an activity at most has a parent activity that is adjacent to and ahead of it on the time line; second, for a message relation, SEND and RECEIVE activities always come in pairs.

For an individual request, it is clear that correlating a causal path is the course of building a CAG with interaction activities as the input. Fig. 3 shows an example of an individual CAG.

A single causal path can help administrators get microlevel user request behavior of services. For example, administrators can detect transient failures of nodes if some causal paths show abnormal information. However, causal paths cannot be directly utilized to represent macrolevel performance signature data of services for two reasons. First, there are massive causal paths. An individual causal path only reflects how a request is served by services. Considering disturbance in environments, we cannot take an individual causal path as a service's performance signature data. Second, different types of requests would produce causal paths with different features. Thus, we propose a *macrolevel abstraction* to represent performance signature data of multitier services.

We propose to classify causal paths into different causal path patterns according to the shapes of CAGs. On the basis

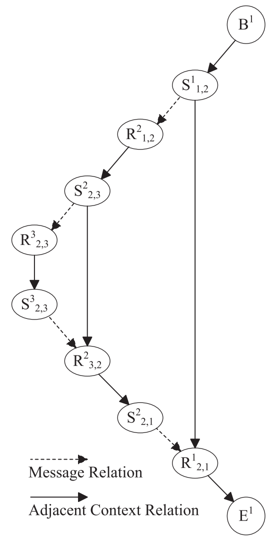


Fig. 3. An example of an individual CAG. The notations are defined in Section 2.

of this abstraction, we further consider which ones are dominated causal path patterns according to their fractions in terms of their path numbers, respectively. Two CAGs will be classified into the same causal path pattern when they meets the following criteria:

1. There are the same number of activities in the two CAGs.
2. Two matching activities with the same order in the two CAGs have the same attribution in terms of *(activity type, program name)*.

In a CAG, for each activity, we define its order based on the following rules:

Rule 1: If $x \rightarrow_c y$ or $x \rightarrow_m y$, then $x \prec y$;

Rule 2: If $x \rightarrow_c y$ and $x \rightarrow_m z$ and there is no relation between y and z, then $x \prec z \prec y$;

Rule 3: If $y \rightarrow_c x$ and $z \rightarrow_m x$ and there is no relation between y and z, then $y \prec z \prec x$.

Rule 1 is obvious. Rules 2 and 3 can be derived from our presumption in Section 2.1. In Section 2.1, we presume that a single execution entity (a process or a kernel thread) of each component can only serve one request in a certain period, which implies a blocking communication mode. For example, in Rule 2, for $x \rightarrow_m z$ and $x \rightarrow_c y$, x must be a SEND activity, and z must be a RECEIVE activity. As an adjacent activity of x, y must happen after z, since the single execution entity under which x happens only triggers y after the return from the call to the execution entity under which z happens.

CAGs include rich performance data of services, because a CAG indicates how a client request is served by each component of a service. For example, if administrators want to pinpoint the performance bottleneck of a service, they need to obtain the service time consumed on each component in serving requests. According to a CAG, we can compute the latencies of components in serving an individual request. For example, for the request in Fig. 1, the latency of *process 2* is $(t(S_{2,3}) - t(R_{1,2}))$, and the interaction latency from *process 1* to *process 2* is $(t(R_{1,2}) - t(S_{1,2}))$, where t is the local time stamp of each activity. The latency of *process 2* is accurate, since all time stamps are from the same node. The interaction latency from *process 1* to *process 2* is inaccurate, since we do not remedy the clock skew between two nodes. For a request, the *server-side latency* can be defined as the time difference between the time stamp of BEGIN activity and that of END activity in its corresponding causal path. For each tier, its role in serving a request can be measured in terms of *the latency percentage of each tier*, which is the ratio of the accumulated latencies of the tier to the server-side latency. The latency percentage of the interaction between two tiers can be defined as the ratio of the accumulated latencies between two tiers to the server-side latency. We adopt the terminology of *latency percentages of components* to describe both latency percentages of each tier and latency percentage of interactions.

After the classification, we can compute the average performance data about causal path patterns, i.e., latency percentage of each tier and latency percentage of interactions, on a basis of which we can further detect performance problems of multitier service or provide online performance data for the feedback controller that aims to save cluster power consumption. Our experiments in Appendix B.3, available in the online supplemental material, and our work in power management [32] demonstrate the effectiveness of this approach in debugging performance-in-the-large and saving cluster power consumption, respectively.

3.3 Tracing Algorithm

Before we proceed to introduce the algorithm of Ranker, we explain how Engine stores incomplete CAGs. In the course of building CAGs, all incomplete CAGs are indexed with two index map data structures. *An index map* maps a key to a value, and supports basic operations, like search, insertion, and deletion. One index map, named *mmap*, is used to match message relations, and the other one, named *cmap*, is used to match adjacent context relations. For mmap, the key is the message identifier of an activity, and the value of mmap is an unmatched SEND activity with the same message identifier. The key in cmap is the context identifier of an activity, and the value of cmap is the latest activity with the same context identifier.

In the following, Section 3.3.1 explains how to choose candidate activities in constructing CAGs in our algorithm; Section 3.3.2 introduces how to construct CAGs, and Section 3.3.3 describes how to handle disturbances.

3.3.1 Selection of Candidate Activities for Composting CAGs

For each service node, we choose the minimal local time stamp of activities as the initial time. We set a *sliding time window* for processing the activity stream. Activities, logged on different nodes, will be fetched into the buffer of Ranker if their time stamps are within the sliding time window. Section 3.3.3 will present how to deal with clock skews in distributed systems.

Ranker puts each activity into several different queues according to the IP address of its context identifier. Naturally, activities in the same queue are sorted according to the same local clock, so Ranker only needs to compare head activities of each queue, and selects candidate activities for composing CAGs based on the following rules:

Rule 4: If a head activity A in a queue has the RECEIVE type and Ranker has found an activity X in the mmap, of which $X \rightarrow_m A$ holds true, then A is the candidate.

If a key is the message identifier of an activity A and the value of the mmap points to a SEND activity X with the same message identifier, we can say $X \rightarrow_m A$.

Rule 4 ensures that when a SEND activity has become a candidate and been delivered to Engine, the RECEIVE activity having message relation with it will also become a candidate once it becomes a head activity in its queue.

Rule 5: If no head activity is qualified with Rule 4, then Ranker compares the type of head activities in each queue according to the priority of $BEGIN \prec SEND \prec END \prec RECEIVE \prec MAX$. The head activity with the lower priority is the candidate.

Rule 5 ensures that a SEND activity X always becomes a candidate earlier than a RECEIVE activity A, if $X \rightarrow_m A$ holds true.

After a candidate activity is chosen, it will be popped out from its queue and delivered to Engine, and Engine matches the candidate with an incomplete CAG. Then, the element next to the popped candidate will become a new head activity in that queue. At the same time, Ranker will update the new minimal time stamp in the sliding time window, and fetch new qualified activities into the buffer of Ranker.

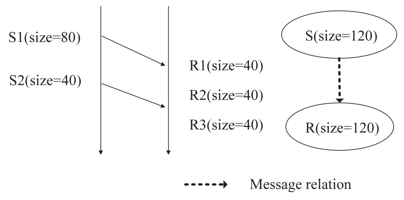


Fig. 4. Merging multiple SEND and RECEIVE activities.

3.3.2 Constructing CAG

Engine fetches a candidate, outputted by Ranker, and matches it with an incomplete CAG. In Appendix A, available in the online supplemental material, the pseudocode illustrates the correlation algorithm. In line 1, Engine iteratively fetches a candidate activity *current* by calling function *rank()* of Ranker. From lines 2-37, Engine parses, and handles activity *current* according to its activity type. Lines 3-11 handle BEGIN and END activities. For a BEGIN activity, a new CAG is created. For an END activity, the construction of its matched CAG is completed.

Lines 12-37 handle SEND and RECEIVE activities. Activities are inherently asymmetric between a sender and a receiver because of their underlying buffer sizes and delivery mechanisms. Thus, a match between SEND and RECEIVE activities is not always one-to-one, but *n-to-n* relations. Fig. 4 shows a case in which a sender consecutively sends a message in two parts and a receiver receives messages in three parts. Our algorithm correlates and merges these activities according to the message sizes in the message identifiers. If some RECEIVE activities are lost, the received message size will be less than the sent message size, but this would not prevent the algorithm from constructing a CAG.

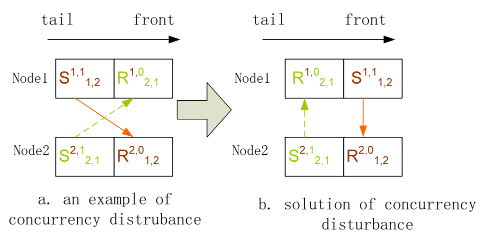


Fig. 5. An example of concurrency disturbance.

It is possible that an activity is wrongly correlated to two causal paths, because of reusing threads in some concurrent programming paradigms. For example, in a thread-pool implementation, one thread may serve one request at a time; when the work is done, the thread is recycled into the thread pool. Lines 31-33 check if two parents are in the same CAG. If the check returns true, Engine will add an edge of a context relation.

3.3.3 Disturbance Tolerance

In an environment without disturbance, our algorithm can produce correct causal paths. However, in practice, there are many disturbances. In the rest of this section, we consider how to resolve noise activities disturbance, concurrency disturbance, and clock skew disturbance.

Noise activities disturbance. Noise activities are caused by other applications coexisting with the target service on the same nodes. Their activities through the kernel's TCP stack will also be logged and gathered by our tool. Ranker handles noise activities in two ways: 1) it filters noise activities according to their attributes, including program name, IP and port; and 2) If activities cannot be filtered with the attributes, the ranker checks them with *is_noise()* function. If true, the ranker will discard them. The pseudocode of *is_noise()* function can be found at Appendix B, available in the online supplemental material.

Concurrency disturbance. The second disturbance is called *concurrency disturbance*, which only exists in multiprocessor nodes. Fig. 5a illustrates a possible scenario, of which two concurrent requests are concurrently served by two multiprocessor nodes and four activities are observed. $S^{1,1}_{1,2}$ means a SEND activity produced on the *CPU1* of *Node1*, and $R^{2,0}_{1,2}$ is its matched RECEIVE activity produced on the *CPU0* of *Node2*. When these four activities are fetched into the buffer of Ranker, they are put into two queues as shown in Fig. 5a. The head activities of both two queues are RECEIVE activities, and hence they block the matched SEND activities of each other. This case is detected according to two conditions: 1) both head activities of two queues are RECEIVE activities. 2) SEND and RECEIVE activities in two queues are matching with each other, respectively. Ranker handles this case by swapping the head activity and its following activity in the first queue. Fig. 5b illustrates our solution.

Clock skew disturbance. As explained in Section 3.3.1, activities will be fetched into the buffer of Ranker according to their local time stamp. However, due to clock skew, RECEIVE activities might be fetched into the buffer before their corresponding SEND activities. We take a simple solution to resolve this issue. Comparing head activities of each queue, we record the time stamp of the first activity from each node. Then, we compute the approximate clock skew between two nodes. Based on the approximate clock skew, we remedy the time stamp of activities on the node with the larger clock skew, and hence we can prevent the scenario mentioned above from happening.

3.4 Improving the System Scalability

We use two mechanisms, tracing on demand and sampling, to improve the system scalability.

3.4.1 Tracing on Demand

The instrumentation mechanism of PreciseTracer depends on a open source software named *SystemTap* [33], which extends the capabilities of *Kprobe* [31]—a tracing tool on a single Linux node. Using SystemTap, we have written the *LOG_TRACE* module, which is a part of Gather. LOG_TRACE obtains context information of processes and threads from the operating system, and further inserts probe points into *tcp_sendmsg()* and *tcp_recvmsg()* functions of the kernel communication stack to log sending or receiving activities.

Deployed on each node, Gather receives commands from Coordinator. When PreciseTracer is enabled or disabled on user demand, Coordinator will synchronize each Gather to dynamically load or unload the kernel module LOG_TRACE, which is supported by the Linux OS. The instrumentation mode of PreciseTracer can be set as *continuous collection*, *tracing on demand*, or *periodical sampling*. When administrators detect the running states of services are abnormal, they can choose the mode of tracing on demand, in which PreciseTracer starts tracing requests according to the commands of administrators. When administrators have pinpointed problems, they can stop tracing requests. When PreciseTracer is set as the mode of periodical sampling, it will be enabled and disabled alternately, which reduces the overhead of PreciseTracer and improves the system scalability.

3.4.2 Sampling

Sampling is a straightforward solution to reduce the amount of logs produced by tracing requests of multitier services. However, it is not a trivial issue to support sampling in precise request tracing approaches. First, the tracing mechanism should be flexible enough to be enabled or disabled on demand. Second, the tracing algorithm must tolerate log losses. In the following, we discuss how to tolerate losses of activities and consider three different cases:

- **Case 1**: Lost BEGIN and END activities;
- **Case 2**: Lost RECEIVE activities;
- **Case 3**: Lost SEND activities.

It is difficult to handle Case 1. Each CAG needs a BEGIN activity and an END activity to identify its begin and end. Fortunately, losses of BEGIN and END activities only affect the construction of their affiliated CAG, and have no influence on other CAGs whose BEGIN and END activities have been identified.

About Case 2, due to the underlying delivery mechanism, a receiver will receive a message in several parts, which is mentioned in Section 3.3.2. The situation that all parts of a message fail to be collected seldom happens.

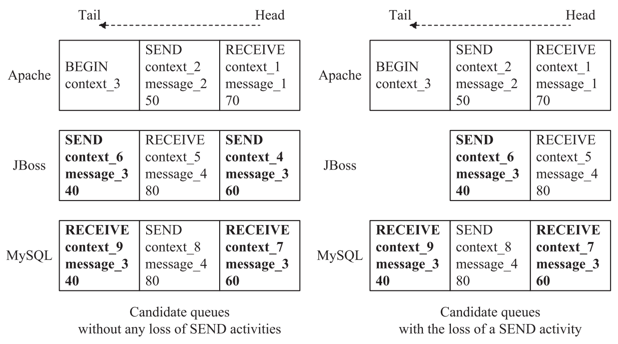

Tail
Head
Apache
BEGIN context_3
SEND context_2 message_2 50
RECEIVE context_1 message_1 70
JBoss
SEND context_6 message_3 40
RECEIVE context_5 message_4 80
SEND context_4 message_3 60
MySQL
RECEIVE context_9 message_3 40
SEND context_8 message_4 80
RECEIVE context_7 message_3 60
Candidate queues without any loss of SEND activities
Tail
Head
Apache
BEGIN context_3
SEND context_2 message_2 50
RECEIVE context_1 message_1 70
JBoss
SEND context_6 message_3 40
RECEIVE context_5 message_4 80
MySQL
RECEIVE context_9 message_3 40
SEND context_8 message_4 80
RECEIVE context_7 message_3 60
Candidate queues with the loss of a SEND activity

Fig. 6. A case of lost SEND activities.

Therefore, in Case 2, the *received message size* will be less than its corresponding *sent message size*, but this wouldn't prevent us from constructing a CAG.

For Case 3, Fig. 6 shows a scenario of lost SEND activities when candidate activities are in queues of Ranker. Activities from the same node are put into a queue and ordered based on their local time stamps. We hereby utilize *(activity type, context identifier, message identifier, message size*) to identify an activity. In Fig. 6, (SEND, context_4, message_3, 60) is related to (RECEIVE, context_7, message_3, 60), while (SEND, context_6, message_3, 40) is related to (RECEIVE, context_9, message_3, 40) and they share the same message identifier. Ranker would pick candidate activities. However, if it fails to collect the activity of (SEND, context_4, message_3, 60), the activity types of all head activities will be RECEIVE, and our algorithm mentioned above cannot proceed. We take a simple solution to resolve this issue, and we just discard the RECEIVE activity with the smallest time stamp. In Appendix B.1, available in the online supplemental material, our experiments show that our solutions of handling lost activities perform well.

3.4.3 The Complexity of the Algorithm

For a multitier service, the time complexity of our algorithm is approximately $O(g * p * \Delta n)$, where g measures the structure complexity of a service, p is the number of requests in a certain duration, and Δn is the size of activity sequence per request in a sliding time window. Furthermore, the time complexity of our algorithm can be expressed as $O(g * n)$, where n is the size of activity sequence in a sliding time window. The space complexity of our algorithm is approximately $O(2g * p * \Delta n)$ or $O(2g * n)$.

4 PreciseTracer Implementation

We have implemented PreciseTracer with three key components: *TCP_Tracer, Correlator, Analyzer*.

After the kernel module named LOG_TRACE (which is a part of Gather) is loaded, a logging point will be trapped to generate an activity log whenever an application sends or receives a message. The original format of an activity log produced by LOG_TRACE is *"time stamp hostname program_name Process ID Thread ID SEND/RECEIVE sender_ip: port-receiver_ip: port message_size"*. Gather further transforms the original logs into more informative n-ary tuples to describe the context and message identifier of each activity (described in Section 3.1). Determining activity types is straightforward: SEND and RECEIVE activities are transformed directly; BEGIN or END activities are determined by the ports of the communication channel. For example, the RECEIVE activity from a client to the webserver's port 80 means the START of a request, and the SEND activity via the same connection in opposite direction means END of a request. After all Gathers have finished log transformation, Coordinator will start Correlator and Analyzer to further process the collected logs.

Correlator constructs CAGs and delivers them to Analyzer. Analyzer analyzes CAGs to obtain causal path patterns, and further derives statistical information about those patterns.

5 Conclusion

We have developed an accurate request tracing tool, called *PreciseTracer*, to help users understand and debug performance problems in a multitier service of black boxes. Our contributions lie in fourfold:

1. We have designed a precise tracing algorithm to derive causal paths for each individual request, which only uses application-independent knowledge, such as time stamps and end-to-end communication channels;
2. We have presented two abstractions, component activity graph and dominated causal path pattern, for understanding and debugging microlevel and macrolevel user request behaviors of the services, respectively;
3. We have developed two mechanisms, tracing on demand and sampling, to increase the system scalability; and
4. We have designed and implemented an online request tracing system.

To validate the efficacy of PreciseTracer, we have conducted extensive experiments on 3-tier platforms. In comparison with WAP5—a black-box tracing approach, PreciseTracer achieves higher tracing accuracy and faster response time. Our experimental results also show PreciseTracer has low overhead, and still achieves high tracing accuracy even if an aggressive sampling policy is adopted, which indicates that PreciseTracer is a promising tracing tool for large-scale production systems.

Acknowledgments

The authors are very grateful to anonymous reviewers. This work is supported by the Chinese 973 project (Grant No. 2011CB302500) and the NSFC project (Grant No. 60933003). The work was performed when Bo Sang was with the Institute of Computing Technology, CAS. Jianfeng Zhan is the corresponding author.

References

[1] P. Barham et al., "Using Magpie for Request Extraction and Workload Modeling," *Proc. Sixth Conf. Symp. Operating Systems Design and Implementation (OSDI)*, pp. 18-18, 2004.
[2] P. Barham et al., "Magpie: Online Modelling and Performance-Aware System," *Proc. Conf. Hot Topics in Operating Systems (HotOS '03)*, pp. 85-90, 2003.
[3] P. Reynolds et al., "WAP5: Black-Box Performance Debugging for Wide-Area Systems," *Proc. 15th Int'l Conf. World Wide Web (WWW)*, pp. 347-356, 2006.

[4] M.K. Aguilera et al., "Performance Debugging for Distributed Systems of Black Boxes," *Proc. 19th ACM Symp. Operating Systems Principles (SOSP),* pp. 74-89, 2003.

[5] E. Koskinenand et al., "BorderPatrol: Isolating Events for Black-Box Tracing," *SIGOPS Operating System Rev.,* vol. 42, no. 4, pp. 191-203, 2008.

[6] M. Ricahrd Stevens, *UNIX Network Programming Networking APIs: Sockets and XTI,* vol. 1, Prentice Hall, 1998.

[7] S. Agarwala et al., "E2EProf: Automated End-to-End Performance Management for Enterprise Systems," *Proc. 37th IEEE Int'l Conf. Dependable Systems and Networks (DSN),* pp. 749-758, 2007.

[8] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, 1978.

[9] B.P. Miller, "DPM: A Measurement System for Distributed Programs," *IEEE Trans. Computers,* vol. 37, no. 2, pp. 243-248, Feb. 1988.

[10] B. Tierney et al., "The NetLogger Methodology for High Performance Distributed Systems Performance Analysis," *Proc. 17th Int'l Symp. High Performance Distributed Computing (HPDC),* pp. 260-267, 1998.

[11] J.L. Hellerstein et al., "ETE: A Customizable Approach to Measuring End-to-End Response Times and Their Components in Distributed Systems," *Proc. 19th Int'l Conf. Distributed Computing System (ICDCS),* pp. 152-162, 1999.

[12] E. Thereska et al., "Stardust: Tracking Activity in a Distributed Storage System," *Proc. Joint Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS),* pp. 3-14, 2006.

[13] P. Reynolds et al., "Pip: Detecting the Unexpected in Distributed Systems," *Proc. Third Conf. Networked Systems Design and Implementation (NSDI),* pp. 115-128, 2006.

[14] A. Chanda et al., "Whodunit: Transactional Profiling for Multi-Tier Applications," *SIGOPS Operating Systems Rev.,* vol. 41, no. 3, pp. 17-30, 2007.

[15] M.Y. Chen et al., "Pinpoint: Problem Determination in Large, Dynamic Internet Services," *Proc. 32th Int'l Conf. Dependable Systems and Networks (DSN),* pp. 595-604, 2002.

[16] A. Chanda et al., "Causeway: Operating System Support for Controlling and Analyzing the Execution of Distributed Programs," *Proc. 10th Conf. Hot Topics in Operating Systems (HotOS),* pp. 18-18, 2005.

[17] R. Fonseca et al., "X-Trace: A Pervasive Network Tracing Framework," *Proc. Fourth USENIX Conf. Networked Systems Design and Implementation (NSDI),* pp. 271-284, 2007.

[18] A. Anandkumar et al., "Tracking in a Spaghetti Bowl: Monitoring Transactions Using Footprints," *Proc. Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '08),* pp. 133-144, 2008.

[19] M.Y. Chen et al., "Path-Based Failure and Evolution Management," *Proc. USENIX Symp. Networked Systems Design and Implementation (NSDI '04),* 2004.

[20] Z. Zhang et al., "Precise Request Tracing and Performance Debugging of Multi-Tier Services of Black Boxes," *Proc. IEEE Int'l Conf. Dependable Systems and Networks (DSN '09),* pp. 337-346, 2009.

[21] B. Sang et al., "Decreasing Log Data of Multi-Tier Services for Effective Request Tracing," *Proc. IEEE Int'l Conf. Dependable Systems and Networks (DSN '09),* 2009.

[22] B.C. Tak et al., "vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities," *Proc. Conf. USENIX Ann. Technical Conf. (USENIX '09),* 2009.

[23] B.M. Cantrill et al., "Dynamic Instrumentation of Production Systems," *Proc. Conf. USENIX Ann. Technical Conf. (USENIX '04),* 2004.

[24] Y. Ruan et al., "Making the "Box" Transparent: System Call Performance as a First-Class Result," *Proc. Ann. Conf. USENIX Ann. Technical Conf. (USENIX ATC '04),* 2004.

[25] Y. Ruan et al., "Understanding and Addressing Blocking-Induced Network Server Latency," *Proc. Ann. Conf. USENIX Ann. Technical Conf. (USENIX ATC '06),* 2006.

[26] K. Shen et al., "Hardware Counter Driven on-the-fly Request Signatures," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII),* pp. 189-200, 2008.

[27] B.H. Sigelman et al., "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," Google Technical Report dapper-2010-1, Apr. 2010.

[28] J. Tan et al., "Visual Log-Based Causal Tracing for Performance Debugging of MapReduce Systems," *Proc. IEEE 30th Int'l Conf. Distributed Computing Systems (ICDCS '10),* 2010.

[29] C. Stewart et al., "Performance Modeling and System Management for Multi-Component Online Services," *Proc. Conf. Symp. Networked Systems Design and Implementation (NSDI '05),* 2005.

[30] K. Appleby et al., "Oceano—SLA Based Management of a Computing Utility," *Proc. IFIP/IEEE Symp. Integrated Network Management,* pp. 855-868, 2001.

[31] R. Krishnakumar, "Kernel Korner: kprobes-a Kernel Debugger," *Linux J.,* vol. 2005, no. 133, p. 11, May 2005.

[32] L. Yuan et al., "PowerTracer, Tracing Requests in Multi-tier Services to Save Cluster Power Consumption," technical report, http://arxiv.org/corr/, 2010.

[33] SystemTap, http://sourceware.org/systemtap, 2011.

[34] TPC Benchmark, http://www.tpc.org/tpcw/, 2011.

[35] G. Ren et al., "Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers," *IEEE Micro,* vol. 30, no. 4, pp. 65-79, July 2010.

[36] P. Wang et al., "Transformer: A New Paradigm for Building Data-Parallel Programming Models," *IEEE Micro,* vol. 30, no. 4, pp. 55-64, July 2010.

[37] L. Wang et al., "In Cloud, Do MTC or HTC Service Providers Benefit from the Economies of Scale?" *Proc. Workshop Many-Task Computing on Grids and Supercomputers (MTAGS '09),* 2009.

[38] L. Wang et al., "In Cloud, Can Scientific Communities Benefit from the Economies of Scale?," *IEEE Trans. Parallel and Distributed Systems,* vol. 23, no. 2, pp. 296-303, Feb. 2012.

[39] Readings in Instrumentation, Profiling, and Tracing, http://www.inf.usi.ch/faculty/hauswirth/teaching/ipt.html, 2011.

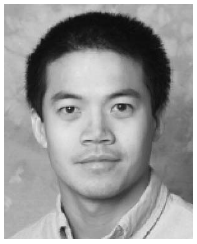
[40] X. Liu et al., "Automatic Performance Debugging of SPMD-Style Parallel Programs," *J. Parallel and Distributed Computing,* vol. 71, no. 7, pp. 925-937, July 2011.

Bo Sang received the bachelor's degree in computer science from Nanjing University, China, in 2007. Under the direction of professor Jianfeng Zhan, he received the master's degree in computer science, from Institute of Computing Technology, Chinese Academy of Sciences, China, in 2010. He is currently working toward the PhD degree in the Department of Computer Science, Purdue University. He joined Purdue University in 2010 and work with professor Dongyan Xu in FRIENDS research group. Currently, his research interests include reliability and dependency of distributed system and cloud computing.

Jianfeng Zhan received the PhD degree in computer engineering from Chinese Academy of Sciences, Beijing, China, in 2002. He is currently an associate professor of computer science with Institute of Computing Technology, Chinese Academy of Sciences. His current research interests include distributed and parallel systems. He has authored more than 40 peer-reviewed journal and conference papers in the aforementioned areas. He is one of the core members of the petaflops HPC system and data center computing projects at Institute of Computing Technology, Chinese Academy of Sciences. He was a recipient of the Second-class Chinese National Technology Promotion Prize in 2006, and the Distinguished Achievement Award of the Chinese Academy of Sciences in 2005.

Gang Lu received the BS degree in 2010 from Huazhong University of Science and Technology in China, in computer science. He is currently working toward the PhD degree in computer science at the Institute of Computing Technology, Chinese Academy of Sciences. His research focuses on parallel and distributed computing.

Haining Wang received the PhD degree in computer science and engineering from the University of Michigan at Ann Arbor in 2003. He is an associate professor of computer science at the College of William and Mary, Williamsburg, VA. His research interests lie in the area of networking systems, security, and distributed computing. He is a senior member of the IEEE.



Dongyan Xu received the BS degree from Zhongshan (Sun Yat-Sen) University in 1994 and the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 2001. He is an associate professor of computer science and electrical and computer engineering (by courtesy) at Purdue University. His current research areas include virtualization technologies, computer malware defense, and cloud computing. He is a recipient of the US National Science Foundation (NSF) CAREER Award.

Lei Wang received the master's degree in computer engineering from Chinese Academy of Sciences, Beijing, China, in 2006. He is currently a senior engineer with Institute of Computing Technology, Chinese Academy of Sciences. His current research interests include resource management of cluster and cloud systems. He was a recipient of the Distinguished Achievement Award of the Chinese Academy of Sciences in 2005.

Zhihong Zhang does not wish to have a photograph or biography published.

Zhen Jia received the BS degree in 2010 from DaLian University of Technology in China, in computer science. He is currently working toward the PhD degree in computer science at the Institute of Computing Technology, Chinese Academy of Sciences. His research focuses on parallel and distributed computing.